Computer Science Department

TECHNICAL REPORT

THE DESIGN AND ANALYSIS OF A DISTRIBUTED PROCESSING SYSTEM

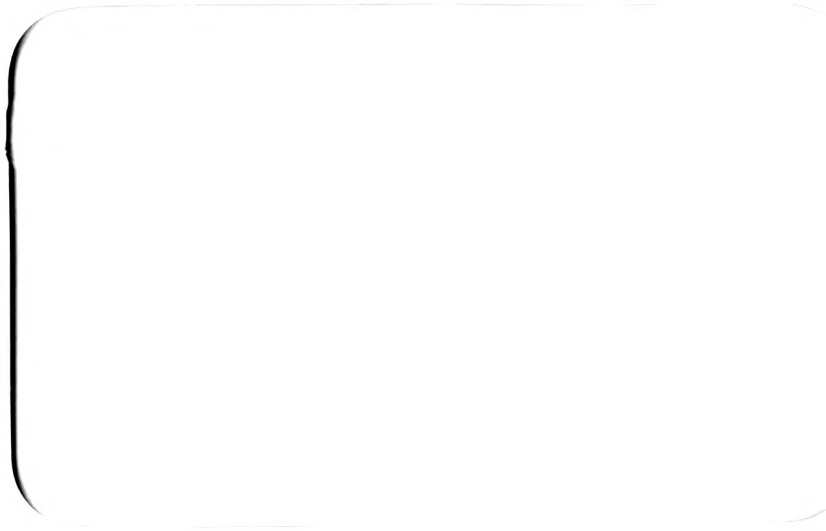
By

Andrew Green

February 1985
Technical Report #152

NEW YORK UNIVERSITY

Department of Computer Science
Courant Institute of Mathematical Sciences
251 Mercer Street, New York, N.Y. 10012



THE DESIGN AND ANALYSIS OF A DISTRIBUTED PROCESSING SYSTEM

By

Andrew Green

February 1985
Technical Report #152

The Design and Analysis of a Distributed Processing System

By Andrew Green

February 1985

A thesis in the Department of Computer Science submitted to the faculty of the Graduate School of Arts and Science in partial fulfillment of the requirements for the degree of Master of Science at New York University.

Approved By

Paul C. Spirakis

Paul Spirakis

Design and Analysis of a Distributed Processing System

Andrew Green
Courant Institute of Mathematical Sciences

ABSTRACT

The design of a distributed operating system based on a small set of synchronous message passing primitives, send-receive-reply, is explored. A description of how the UNIX† operating system can be extended by these primitives to support a local area network environment consisting of diskless workstations connected by Ethernet to remote file servers is given. A performance analysis of the consequences of having all files transferred remotely over the network is presented. It is concluded that the proposed distributed UNIX is competitive with traditional time-sharing operating systems.

Introduction

This thesis is divided into two parts. In the first part, a concise message passing methodology is introduced and explored. The send-receive-reply message primitives are the basis of this methodology and allows for a client-server model of interprocess communication. Their use in Thoth [Cheriton 79], a real-time operating system, and V [Cheriton 83], a distributed version of Thoth, is examined. Thoth and V implement a generalization of a server process, known as an *administrator*, which provides a well organized process-structured solution to the client-server model. The ideas of Thoth and V are then applied to the UNIX [Ritchie 74] operating system to explore the possibility of a distributed version of UNIX. Specifically, the V kernel architecture of single user diskless work stations connected by Ethernet [Metcalfe 76] to remote file servers is the basis for our distributed UNIX architecture. This implies that the UNIX file system, formerly a kernel service, is now distributed to a remote processor as a server process. The consequences of

† UNIX is a Trademark of Bell Laboratories

remote file servers for UNIX are discussed. The decision to provide a message interface, based on V's primitives, for user applications rather than a procedural interface for UNIX is explained and justified. Finally, the UNIX kernel is then shown to be adaptable to changes to provide message passing primitives. Although UNIX is a time-sharing system, it is concluded that the approach to message passing of the V kernel can be exploited by UNIX.

In the second part of this thesis, the implications of the unconventional V kernel architecture is analyzed. There is some empirical evidence to support the claim that the V kernel provides good system performance. However the evidence does not indicate what the average behavior of this architecture would be like for the UNIX environment. Specifically, we are interested in the behavior of this system in a multi-user environment where most of the network traffic is the transfer of large data segments. This would simulate UNIX programs being loaded from the file server. The Buzen central server model is the basis for our analysis. The results of this analysis will be used to determine how many workstations a given configuration of the system can adequately be supported.

Table of Contents

Design of a Distributed System

1. Message Passing Methodology

In designing a distributed operating system, an important decision that must be made is the method of interprocess communication (ipc.). Generally, if the architecture is based on multiple processors sharing a common memory, then semaphores are used. However, if processors communicate through a bus or a communication line, then message passing becomes more practical. Lauer and Needham [Lauer 78] have shown that message passing and semaphores have equivalent computational efficiency in shared memory systems. But, in truly distributed systems, where processors are physically separated, semaphores are an inefficient use of the communications channel. In these types of systems, message passing is a more practical alternative. It, also, has other attractions besides efficiency, not the least of which is its intuitive appeal. The semantics of message passing , though, are varied depending on the particular application to which they are being applied. I would like to explore a particular methodology of message passing and show that is well suited for a single processor operating system and that of a distributed processor operating system under certain restrictions.

The importance of having well structured constructs, which are restricted in an intelligent manner, has proven to be a successful approach in accommodating the synchronization and communication of concurrent processes. For example, semaphores are constructs which are too unstructured for the writer of concurrent programs to be sure he is producing reliable code. Although it is adequate in some applications, a more useful approach is to use semaphores (or even a simpler locking mechanism) to build a *monitor* into a higher level language [Hoare 74]. Message passing has been used without language support in a number of operating systems (e.g., LOCUS [Popek 81], Accent [Rashid 81], MERT [Lycklama 78], Medusa [Ousterhout 80], Demos [Baskett 78]). However, its uses are not as well understood as monitors. Gentleman [Gentleman 81] addresses this issue by presenting a restricted set of message passing primitives and a powerful model of communicating processes.

Gentleman's approach was based on the experience of the designers of a portable real-time operating system, Thoth. An important goal in Thoth was to

structure programs as multiple processes. This was considered essential in a real-time operating system since a program responds to a variety of asynchronous events. It was felt that a team of processes sharing a common address space was the best way to accommodate this problem: a process for each event. This choice of multiple process structuring determined the type of message passing primitives which were to be used. In Thoth, message passing is synchronized in that a sender of a message must wait until a receiver receives the message *and* replies back. A receiver of a message must, also, wait for a sender. By taking this approach, Thoth restricts the asynchronousness of communication to being between processes rather than within a process. Having the asynchronousness appear in one form allows for more understandable programs even when many processes were involved.

The ideas of Thoth can be extended to time-sharing systems and distributed operating systems. In the first part of this thesis this methodology will be discussed, and a set of message passing primitives will be presented as a way to extend the UNIX operating system for a distributed environment.

Gentleman presents 4 issues involved in the semantics of message passing and suggests why Thoth's approach is valid. These four issues are as follows:

1. blocking vs. non-blocking primitives.
2. addressing of processes.
3. message format.
4. communication failure.

I would like to briefly explore each of these issues along the lines of Gentleman's argument to motivate the Thoth approach to message passing.

1.1. Blocking vs. Non-blocking Primitives.

Message passing primitives must satisfy the two requirements of interprocess communication: synchronization and communication. Blocking primitives successfully satisfy both of these needs. A process blocks when it sends a message until a receiving process has issued a receive primitive. Correspondingly, a process blocks on a receive primitive until a message is sent to it. This approach to message passing is based on Hoare's Communicating Sequential Processes (CSP) notation and can be shown to be as powerful as semaphores [Hoare 78]. Therefore, a

separate mechanism for synchronization is not required.

There are important implementation advantages to blocking primitives that make them appealing on a practical level. With blocking primitives, there is no need for the kernel to dynamically allocate messages from a buffer pool. Therefore, buffer exhaustion is not a problem. However, in a distributed architecture, even with blocking primitives, a buffer pool must be maintained. The problem with blocking primitives, though, is that there is a loss of concurrency in certain instances. For example, a process may block to send a message which is not critical for synchronization but is needed strictly for communication. This can occur after a process has received a message requiring it to perform some service for the requesting process. When the work is completed, the process must merely notify the sender that it is finished so that it can process the next request. With blocking primitives, this process must wait for the sender to be scheduled and is prevented from performing any other work.

There is clearly a need for a non-blocking primitive which would increase concurrency. They come in two varieties. The first type allows a sender to send many messages which are buffered internally for the receiver. The second type allows for a process to return immediately from an attempt to send a message with a success or failure condition depending on whether a process which is expecting the message is already blocked waiting to receive it. The problem with the first type, known as a queuing send is that a process may exhaust the buffer pool. This can be partially solved by putting a bound on the number of messages a process can have outstanding, but this merely hides the problem. The second type essentially involves polling and reintroduces the problem of inadequate concurrency.

There is a third type of non-blocking primitive which the developers of Thoth have used to great advantage and it is known as the reply. The assumption with a blocking send primitive is that the process is unblocked as soon as the intended receiving process issues a receive primitive. However, it is possible to have the sender blocked until the receiver issues a special non-blocking reply primitive. The reply message, issued by the reply primitive, is written into the user data space which has been allocated for the message which had been originally sent. Thoth provides this primitive to allow for the common situation where a process requires some kind of confirmation before it can proceed. The beauty of this approach is

two-fold. For one, it elegantly solves the synchronization problem since the sender of the message is blocked until the receiver (usually called a server) decides to allow it to continue based on state information it is maintaining. Secondly, since every reply implies that some process is blocked on a send, no internal message buffering need be maintained. In practice, it is more efficient to have a kernel message buffer associated with each process to temporarily hold a message. This avoids the problem of having to swap a process back into main memory just to access the message in its address space. This send-receive-reply sequence implies a server oriented view of message passing. That is, there exists a process which controls access to some resource and exclusively through which processes or clients make their requests. I believe this to be a useful approach in designing operating systems and especially distributed operating systems. The client-server model will be discussed later, along with ways it can be refined and its differences with the customer oriented view implied by monitors.

There is a potential problem in message-based real-time operating systems where one process (the server) is responsible for controlling access to a resource. This problem is the loss of the real-time property. Unfortunately, this can occur since the semantics of send-receive-reply imply that messages are received in FIFO order. Typically, a process will inform the server when a resource becomes available by sending it a message. This is an important message since the server is maintaining the state of the system and should know of a state change as soon as possible. However, this important message might then be place at the end of a long queue of messages which originated from other client processes of the server. The server will find out about the new state of the system only after all the preceding messages have been received. In this situation performance can be degraded. It can also be the cause of a failure if the message sent requires the server to take immediate action.

A way to partially avoid this problem is through the use of the receive-specific form of receive. In this case, the receiver would specify the process from which it expects to receive a message and is blocked until that message arrives. The server can then keep a count of the number of messages it has received between notification messages of the freed resource. If a pre-determined number of messages has been received without receiving the notification message, the server

issues a receive-specific the next time it needs a message. Thus, it can guarantee to receive the message within a bounded period of time assuming two conditions. The first is that the notification message can be expected to arrive within a pre-determined period of time. This is true of many resources. A disk will be free for another I/O transfer within a known period of time. A clock , of course, has this property. The second condition is that the server process must be able to process a message within a bounded period of time. This is possible but requires a process structuring technique which will be explored later in this thesis. Thus, it is possible to guarantee the real-time property, even under unfavorable conditions for a message passing system based on send-receive-reply.

1.2. Addressing

There are two major problems in addressing. The first is whether processes involved in communication should be explicitly identified as part of the semantics of message passing primitives. The second is how to obtain and use the name of communicating processes. If the communication between processes can be established at process creation time, then the processes may not have to be identified as part of the message passing primitive. A good example of this is a *pipe* between two processes in the UNIX operating system. However, the problem with *pipes* and similar mechanisms is that it does not allow general interprocess communication between unrelated processes.

If the client-server model is to be our goal, then we must specify the communicating processes in our primitives. In a send primitive this is a necessity. Although it is possible to consider a broadcasting send (where all processes receive the message), Gentleman has found this type of send to be of less use. What about specifying the communicating process in the receive primitive? If we are forced to specify a particular process from which we are to receive messages we could be inhibiting concurrency. This is especially true for a server process which does not know ahead of time the identity of its client processes. Therefore, the send primitive should specify the process identity of the receiver while the receiver should be allowed to receive from any process. However, there are uses for a receive primitive which specifies a particular process, as we have seen. This so called receive-specific form of receive should be provided along with the general

receive to allow the user greater choice in his writing of concurrent programs. Note: the use of receive-specific with send is essentially Hoare's CSP.

The reply primitive mentioned above should require the identity of the process which is expecting it. This allows a server process to unblock waiting processes based on its own rules in guarding the resource it controls. If, however, the server must reply to processes in the order in which it receives messages then it is essentially simulating the rendezvous mechanism of Ada. Ada does allow out of order processing, to some extent, through the nesting of **accept** statements [Buhr 84]. However, because the rendezvous's must be terminated in the reverse order, we have succeeded in only stacking requests. This approach, although useful in some applications, prevents the server from issuing reply's in an order of its choosing.

The last issue to be discused is that of determining and using the identity of the processes involved in message passing. If the process identity must be determined at compile time, as with Hoare's CSP, then it becomes impossible to communicate with dynamically created processes. However, a process can be referenced by a unique identifier which is returned as part of the semantics of process creation. This so called pid (process identifier) can be used in all subsequent transactions. An alternate approach is to associate a channel or link identifier with a communicating process. This identifier is, typically, a small integer which is unique on a process basis and is used in all communication, instead of the pid. The assumption in this case is that there is substantial overhead in establishing communication (e.g., to insure that processes are allowed to communicate). The Accent kernel is a good example of this since processes communicate by specifying a port number.

An Accent port is a protected kernel object and the basic transport abstraction. An Accent process is created with two ports which allow it to send and receive messages to the kernel. An Accent process can communicate with another process by first establishing communication with a known server process through one of its kernel ports. When a connection is made with the server, two new ports are created which corresponds to the channel between the server and client. The server then attempts to establish a pair of communication ports with the intended process. Then, through a mechanism that allows access privileges to a port to be passed to other processes, the server process allows the two communicating processes to

access its newly created ports. Thus, a full duplex connection is established between processes. This involved protocol is necessary in an environment where processes are careful about who they communicate with. However, if we assume a friendly environment or one where there is a dedicated application, then using a pid as an address is a satisfactory solution. It should be kept in mind, though, that pid's can not be guaranteed to be unique over time. It is essential that the pid space is large enough to insure that this problem is rare and that a process's anonymity is protected from accidental (or malicious) communication.

1.3. Message Format

The message format is essentially a syntactic problem. The first possibility is to treat communication between processes as a remote procedure call. In this case, the send primitive must specify all the parameters of the *call* while the receiver must determine the appropriate *declaration* header in which to refer to the appropriate fields of the message. The second possibility is to treat the message as a variant record. In this case the receiver is allowed to pass back a different message than has been sent to it. Thoth has taken the second approach. This was felt to allow more flexibility and understandability in message handling. That is, instead of having to match corresponding send and receive arguments one must merely reference the same record structure.

Another question which has to be answered is determining the size of a record. Fixed size messages have an obvious implementation advantage over variable size messages. However, if messages are assumed to be of a fixed size, how does one transfer large amounts of data between processes? This is the so called *large message* problem. It might seem that one possible way to send a variable size message is to packetize it as several fixed size messages. Presumably, the receiver issues a receive-specific and reassembles the large message. The problem with this is that there is no efficient mechanism, in the current scheme, to packetize replies: a reply for each send is too expensive a procedure.

A separate mechanism can be used to transfer large variable size messages, although, making this separation might seem an unreasonable approach. However, it is possible to model message traffic based on the assumption that most communication is of many small messages followed by a large exchange of data.

Thoth takes this approach and has a fixed size message of 8 words to be used in its send-receive-reply primitive. It provides a separate primitive to transfer data segments directly between process address spaces and it is known as a **transfer**.

In Thoth, one can transfer large segments between processes which are waiting for a reply message from the transferring process, including allowing a transfer between this process and one of the blocked processes. The assumption here is that the original send message contains the control information (read or write access, segment address, segment size) which is used by the transferring process to perform the exchange. A reply message is eventually sent to complete the transaction and to indicate the actual amount of data returned. The **transfer** primitive requires a source and destination pid along with the segment address and the size of the message to be transferred. Another possibility is to use variable size messages in the send and receive primitive, but this would requires the kernel to maintain a buffer pool. The **transfer** primitive avoids the problem of maintaining a buffer pool by using direct transfer between address spaces. This is possible since the process issuing the send remains blocked until it receives the reply. If both processes are in core an efficient address space transfer can be used. If the sending process has been swapped out, a disk to user address space transfer must occur.

In a distributed environment, though, we must consider how a primitive like *transfer* would be implemented based on the underlying communication service. In this thesis, we are considering a local area network, Ethernet, and therefore have a packet switching communication service. In V, the distributed version of Thoth, the kernel implements a large message transmission service directly upon a datagram service and avoids using a costly acknowledgement service. This will be described in more detail below.

1.4. Communication Failure

The semantics of send-receive-reply imply a failure in interprocess communication can take several forms. There can be a failure as a result of a deadlock of the processes involved in communication. A process which is expected to reply to a message can be destroyed. Finally, in the case of physically distributed processes a failure can result because of untrustworthy communication. Let us examine each of these cases.

With send-receive-reply, deadlock is possible if a cycle of processes send to each other or block waiting to receive (receive specific) from each other. A sending process is thus requesting a reply message and a receiving process is requesting a send message. Therefore, in this case, a cycle in the standard resource graph is sufficient to show deadlock. The receive general form complicates matters. In this case, a process can be effectively deadlocked if all processes which expect to communicate with it are themselves blocked. Deadlock, in a non-distributed system, can be avoided if the operating system maintains a data structure to represent the resource graph. In the case of receive general, a timeout mechanism can be used to free processes which have been blocked for an unacceptably long period of time. It is possible that as part of the syntax of the send or receive primitive, one could add a timeout parameter. However, if this is done, the user must be very careful of the time scale used. If he is not, he will be inhibiting concurrency by being forced to poll for a message, if his timeouts are premature. The designers of Thoth have included a timeout mechanism, but within the operating system and not under user control. In V, a distributed version of Thoth, the operating system will timeout a send to a remote process, if the reply is not received within a pre-determined period of time. This is a simple but effective way to avoid deadlock in a distributed environment and also the basis of providing reliable delivery of inter-machine messages.

If a process is destroyed before its expected reply message is sent, processes can be effectively deadlocked waiting for a message which will never arrive. In this situation, the operating system must free all processes which are blocked on a send or a receive specific to the deceased process. An appropriate error can be returned to the sender to indicate that the transaction has failed. If the process that is destroyed is a well known server process, then we have the problem of how to announce a new server process that is brought up. Typically, there exists a name server which allows a client process to determine the pid of the server processes. When a server process is brought up, he announces his pid to the name server. This allows prospective clients to determine the new pid by first asking the name server for the server's identity. In Thoth, commonly used system processes (such as a name server, terminal server, file server, etc.) are assigned a logical pid. The kernel maintains a mapping between the logical pid and the actual pid. A process

can always communicate with a name server since there is a system call which returns the pid associated with a logical pid.

In the case of a distributed multiprocessor system, the problem of communication errors usually requires a separate protocol layer (with associated primitives) that guarantee delivery of a message. However, as we will see, the send-receive-reply primitives are sufficient in a local area network environment where error rates are low. Essentially, the reply message is used as both an acknowledgement of delivery and to deliver the actual user message. The designers of a distributed operating system, V, take this approach and base their interprocess communication primitives on Thoth's.

1.4.1. Fault Recovery

An interesting area related to communication failure is that of recovery from error. If, for example, a server process has been destroyed as a result of a hardware or software fault, is it possible to provide continued service without the clients knowledge? Recently, this has been an active area of research and ,in fact, many commercial operating systems are now available which provide some level of fault-tolerance.

The Tandem NonStop kernel [Bartlett 81] is a distributed operating system that uses a model of message passing that is very similar to that of Gentleman's. This is the basis of their approach to fault tolerance. The NonStop kernel provides an environment in which every process has a back-up process executing on another processor. Before a reply message is sent back from the server, the server checkpoints its process state to that of a backup server. The checkpointing includes copying selected portions of the primary's data segment to that of the backup process so that this process will recover with the last checkpointed state of the primary. However, if the server fails before the last checkpoint, it is possible for the last message sent by the client to be reprocessed by the server. In many applications this is not a problem. For example, a terminal server may redraw a screen, and even a file record may be rewritten in the same absolute place in a file [Bartlett 78].

As noted by Russel [Russel 77], when message passing primitives are asymmetric, the amount of state restoration required for processes is limited. That is, send-receive-reply imply a *direction of information flow* between a client and a server. So, if the producer of a message sends it to a consumer and the consumer fails, then only the consumer must be restored to the last checkpointed state (provided the last message was saved in a so-called *recovery cache*). In general, if the producer fails after a message is delivered it may require other processes to be restored to their previous checkpointed states as well. However, Russel notes that by checkpointing carefully one can limit how far back processes must be restored . This prevents the so-called *domino effect* where a failure may cause a set of communicating processes to be unnecessarily restored to their initial state and thereby undo a distributed computation.

2. Multi-process Structuring and the Client-Server Model

The previous section was an attempt to motivate the use of the send-receive-reply primitives. Their explicit syntax is as follows:

```
receivers_id=send(msg,pid)
requestors_id=receive(msg)
requestors_id=receivid(msg,pid)
reply(msg,pid)
```

Their use in Thoth was based on several important assumptions. They assume that most message traffic is the exchange of small amounts of data. They assume that exchanges are short lived, so that there is no need to maintain a channel or link between communicating processes. They assume that many processes, as part of a team sharing the same address space, are cooperating on a particular application. This implies that large message transfer between related processes is very fast. However, in Thoth we find that large message transfer between arbitrary processes is provided by a separate primitive, but the assumption is that this is less likely an occurrence. Finally, they assume that messages are rarely sent to an unintended process.

Thoth was intended as a real-time operating system whose users would be writing, for the most part, dedicated applications. Thoth, for example, was used to control a model train set [Cheriton 82]. Thoth's use of multi-process structuring to build highly concurrent servers, does, I believe, have uses outside of real-time operating systems. Before discussing the type of systems in which this approach would be useful, I would first like to examine the client-server model.

2.1. The Client-Server Model

Concurrent programs can be written either for a message oriented system or a procedure oriented system (using monitors or semaphores). Lauer and Needham [Lauer 78] have shown that there is a duality between these two approaches under the following transformation:

message <-> process
process <-> monitor
send/reply <-> call

However, there are interesting differences between these two schemes. Monitors attempt to hide the existence of other processes. The monitor is merely a collection of procedures which is called by the process when it needs access to a resource. In order for a monitor to protect its shared variables, it must reveal the existence of other processes by providing the mechanisms of *wait* and *signal*. This way of writing concurrent programs is called client oriented in that the client process is doing all the work. In message oriented systems, the identity of processes providing the services are hidden from the client. Also, the client and server are two separate processes. This view is called server oriented. The server provides mutual exclusion and synchronization for its client process, the same as monitors. In addition, message oriented systems can allow for error recovery (as we have seen) more easily than procedure oriented systems. However, when designing a server it is easy to make the mistake of restricting concurrency by allowing a client process to be blocked waiting for a reply that could have been sent sooner. This is essentially the problem with monitors. For with monitors, the temptation is to place all code and data, whether critical or not, inside the monitor construct. This inhibits concurrency. Gentleman notes that this could be a problem with servers and calls

this kind of server a *proprietor*. For example, consider the following proprietor:

```
proprietor()
{
 while(TRUE) {
   requestor= receive(msg);
   /*provide service*/
   reply(result,requestor);
   /*non-critical service*/
}
```

Note that there is little concurrency provided by the server. The server processes one request at a time forcing prospective clients to wait until the non-critical service is performed. Although it is true that a client is unblocked as soon as the critical section is executed, Gentleman, though, has found this concurrency to be inadequate. The server, then, is merely a sub-routine call (an expensive one) of the client and we are simulating a monitor.

Gentleman also speaks of servers as administrators. An administrator attempts to increase concurrency by providing *worker* processes which perform services formerly provided by the proprietor which can occur concurrently. The administrator's job is to queue requests for work for the worker processes, maintain the state information for these processes, and send work to them when they complete. The worker processes issues a send to the administrator when it has completed its work. The administrator, based on its state information, issues non-blocking a reply when it has more work for the workers. The administrator never issues a send. In fact, if it needs to send a message purely for communication purposes, it uses a worker process, known as a *courier* to perform this task. If we instead had the administrator issue a send and the worker a receive-reply the administrator would be blocked and our solution reduces to that of a proprietor. An important idea in this approach is that the administrator is not blocked accept when it is waiting to process the next message.

Let us consider using this structuring technique to design a solution to the problem of writing a program to control a disk. The client is aware only of the identity of a disk controller process. However, the disk controller process really has two worker processes as assistants. One process accepts disk requests from the disk controller and returns the next disk request that should be processed based on a

latency reduction algorithm. The other process initiates the disk I/O and performs any house keeping tasks. The following "C" code is a sketch of these processes:

```
typedef struct req {
  rqtype  request;
  int     pid;
  ioinfo  io;
}req ;
disk_contr()
{
req msg;
queue *us_que, *di_que;
int disk_done, lat_done, requestor,count;

count=MAXMSGS;
for(;;) {
  if(count != 0)
    requestor=receive(&msg);
  else
    requestor=receivid(&msg,DISK); /*receive specific*/

  switch(msg.request) {
    USER_REQ:
      msg.pid=requestor;
      queue(&msg,us_que);
      break;
    DISK_REQ:
      count=MAXMSGS;
      disk_done=TRUE;
      if(msg.pid)
        reply(&msg,msg.pid);
      break;
    LAT_REQ:
      lat_done=TRUE;
      if(msg.pid)
        queue(&msg,di_req);
      break;
  }
  count=count-1;

  if(disk_done && di_que){
    disk_done=FALSE;
    dequeue(&msg,di_que);
    reply(&msg,DISK);
  }
  if(lat_done && us_que){
    lat_done=false;
    dequeue(&msg,us_que);
    reply(&msg,LAT);
  }
 }
}

disk_io()
{
 for(;;) {
  send(DISK_CON,msg)
  /*perform i/o*/
 }
}

lat_red()
{
 for(;;) {
  send(DISK_CON, msg)
  /*latency reduction algo*/
 }
}
```

In the above example, the *disk_contr* process queues requests for worker processes when they are unavailable (i.e., not blocked on a **send**). Therefore, it is allowed to start another request even though the first has not been completed since either a worker process is available to continue processing this request or else the request is queued until such a time. The *disk_io* process is considered by Gentleman to be a *notifier*, since its primary task is to indicate to the administrator the completion of an external event (i.e., a disk transfer). What makes this an attractive idea is the level of parallelism that could be achieved if there were separate processors for the disk administrator and for the latency reduction process and disk process. Then, while the disk process was initiating I/O the next disk request could be determined while the previous disk request was being sent. Hansen [Hansen 78] notes this possibility for concurrent processes in his Distributed Processes approach.

An important goal in this system is not to allow the *disk_io* process to be blocked any longer than is necessary to actually process its notification message. (The disk should be kept as busy as possible.) As mentioned earlier, the real-time nature of such a system can be maintained by having the administrator process keep a count of the number of messages received between notification messages from the *disk_io* process. After a pre-determined number of messages have been received, the *disk_contr* process issues a receive-specific for the *disk_io* process to insure that the message has been received within the given time period. The administrator process is able to process a message within a period of time bounded by some constant since none of its activities is dependent on the number of message currently being processed. The adding and deleting of messages from a queue (which is the *administrator's* major task) can be accomplished in constant time. The task whose time could not be bounded by a constant (the sorting of disk requests for the latency reduction algorithm) has been given to a separate worker process. Thus, multi-process structuring can be used to insure real-time processing.

3. Operating Systems Considerations

What is required for the **send-receive-reply** primitives and multi-processes structuring to be a useful tool in an operating system? The real time requirements of Thoth would make it appear that processes sharing an address space would be necessary towards this end. However, if we consider the less stringent requirements

of time-sharing systems and assume primitives to efficiently transfer large messages between processes, then this last characteristic loses importance. What I believe to be an essential requirement is that the kernel resources that must be committed to a process be relatively cheap. Multi-process structuring assumes that many processes are used in writing a program. In a real-time operating system like Thoth, many applications can naturally be written this way. In a distributed processing system, where there are many processing elements and efficient ipc. primitives, there is also a need for this type of structuring.

In Thoth the kernel resources that must be committed consist of a process descriptor, a team descriptor (shared by all processes in a team) and a file control block for each file opened by the process. The memory requirements for this would be approximately of the order of 100s of bytes. In terms of memory, each new process merely requires a stack whose size is specified as part of the semantics of the create primitive. Thoth processes are considered by Cheriton to be lightweight. If we are to consider extending this technique to time-sharing and distributed operating systems, then this characteristic should be the one that determines whether these primitives would be successful. There is an operating system whose notable feature is just this. It is called UNIX.

4. The UNIX Kernel

UNIX was designed and implemented at Bell Laboratories by D. Ritchie and K. Thompson during 1969-1971. It is a general purpose, multi-user time-sharing operating system. An early version of UNIX was designed for D.E.C's PDP-11 series of machines. Later versions of UNIX have run on the VAX and have been ported to the IBM Series/1 [Jalics 83] machines. UNIX has a small secure kernel whose notable features are:

1. device independent file system
2. hierarchical file structure
3. ability to initiate asynchronous processes

Then UNIX kernel supports the concept of a process. When a system function is required, the process calls the system as a subroutine. At some point, the process becomes a system process and is able to execute code in the kernel. A system

process has a separate stack from a user process. A user process executes in a virtual memory space consisting of text, data and stack segments. When a system process is executing it uses a small segment, known as the system segment, for a stack and to hold data about that process while the process is an active process (e.g., processor registers and UNIX *file descriptors*). The kernel also maintains a process table that holds information about each process while that process is not active. A UNIX process then requires very little kernel resources (the process table entry and system segment) outside of the memory requirements of the process itself. When a process opens a file, though, more kernel resources are required. A small file structure is allocated for each file opened and holds the byte offset into a file. An information node, *I-node*, is allocated (and eventually written to disk) to be shared by all processes referencing the same file and contains permission information, ownership information, and the physical disk address of the file pages. Finally, the kernel maintains a pool of buffers that are shared by all processes and provide a cache mechanism for file I/O.

Thompson notes that UNIX is more of an I/O multiplexor than an operating system. The kernel does not support a file access method, logical records, physical records and many other features that are normally part of an operating system. UNIX is adept at directing I/O between processes, between a process and a file, and between a process and a device in a uniform manner.

4.1. Device Independence

An interesting feature of UNIX is the way UNIX I/O is structured. UNIX separates I/O into block I/O and character or device I/O. The former would include a disk and the latter would include communications lines, line printers, and paper tapes. However, the same UNIX I/O system calls that are used to read/write files (which are implemented on block I/O devices) are used for all character I/O devices. This is possible since a character I/O device is known to the system as a special file. So, to read/write from any device, one first issues an open call, specifying the name of one of these special files, which the kernel then translates to an open for that device by the appropriate device driver. In all UNIX I/O, a UNIX open of a file returns a small integer, known as a *file descriptor*. This file descriptor is used to identify the file for all successive I/O calls.

4.2. Hierarchical File System

The UNIX file system identifies a file by the device on which it resides and the index of a small structure, which provides access information for a file, within a list. This structure is known as an *i-node* and the index as an *i-number*. UNIX builds a logical hierarchy on this flat file system by the creation of a special file which is a directory. The directory contains a list of file names and their corresponding *i-numbers* Since a directory is another type of file, UNIX allows a hierarchy of directories. The user does not know of a file's *i-number*, only the path of directory names in the hierarchy and the name of the particular file. The kernel holds the *i-number* of the current directory and the root directory of the hierarchy in the system segment of the process. Therefore, references to a file can be in terms of the root directory or the current directory. The kernel then must convert a path name into an *i-number* and then use this *i-number* to obtain the *i-node* This is done by using a known *i-number* (the root directory or current directory) to obtain a directory file. The directory file has the *i-number* of the name of the next component of the path name. If this is the last name in the path, then the corresponding *i-node* will allow us to access the file. If it is not, the *i-node* that is used will allow us to obtain the next directory name in the path and the kernel repeats the process.

4.3. Process Creation

UNIX provides a system call to create a new process and it is known as **fork**. A process issuing this call creates a process which is copy of the original or parent process. The new process or child inherits a copy of the data and stack segments of the parent and continues to execute the same code as the parent. The child also shares all files which were opened by the parent.

4.4. UNIX IPC

One notable feature lacking from the original version of UNIX was a general facility for interprocess communication. (However, recent versions of UNIX have accommodated this problem in varying ways.) UNIX does provide sufficient mechanisms for ipc. for closely related processes (processes with a common ancestor). This is accomplished through a mechanism called a *pipe*. A *pipe* is really an open file connecting two processes. A UNIX process can write into the *pipe* at

one end while another process is reading at the other end. The UNIX kernel provides synchronization and buffering for the processes which are using the pipe. A more global facility for synchronization between unrelated processes is provided by the file system. This is done by creating and deleting a known file, which is used to simulate a binary semaphore. The designers of UNIX felt that the addition of semaphore primitives to UNIX was not necessary not be consistent with the UNIX kernel [Ritchie 78]. The most likely reason why this was avoided was because it was uncertain how the UNIX operating system would be used. If UNIX is to be an operating system for a distributed processor system rather then a single processor then different ipc. facilities would be chosen.

However, it is possible to build a more generalized ipc. facility with UNIX. UNIX allows a *pipe* to be named, this is known as a *fifo*. Thus, unrelated processes can issue file opens and communicate as long as they know this name and have permission. This mechanism could be used as the basis of an ipc. facility in a distributed environment. The problem with this approach is that the ipc. is then based on the file system and communication time is constrained by disk access time.

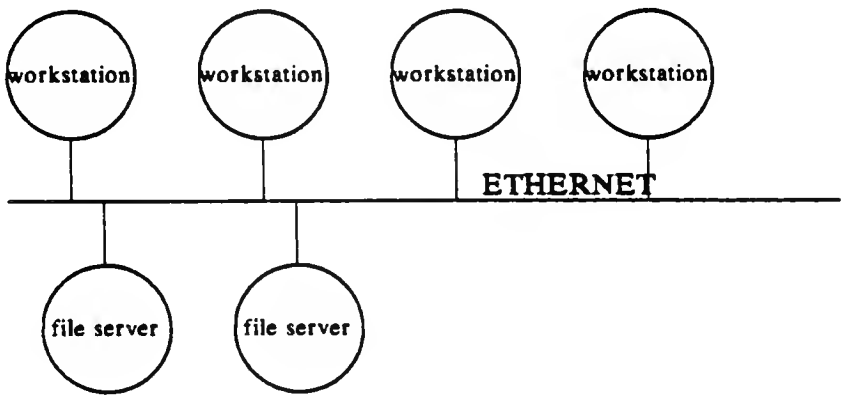
As I have been trying to suggest from the above, a separate ipc. facility is warranted. That is, an ipc. facility separate from the UNIX file system. UNIX would also seem to have the characteristics necessary for Thoth-like primitives. That is, UNIX's most important characteristic is that processes are a cheap resource. However, one desirable characteristic lacking from the original UNIX and important in writing process structured solutions for some applications, is having closely related processes share an address space. However, this is not as important in programs that are not real-time. But, it would be necessary if one were to allow for the capability of implementing a terminal server or a disk server process. These are operating system services that conceivably could be distributed among several processors. The latest release of UNIX from A.T.T. (System V) allows for several processes to share a data segment, so this problem has actually been solved.

Before discussing how these primitives could be used in a distributed UNIX environment, I would like to first present their used in an actual distributed environment. This distributed operating system has several interesting characteristics and will be the basis for our design of a distributed UNIX environment. The most important operating system service it distributes to another

processor, the file system, is moved in its entirety, since the its local operating system is diskless!

5. The V Kernel

The V kernel [Cheriton 83] is a message oriented kernel that is intended for a conventional distributed architecture of workstations connected by a high speed local network to a series of file servers. However, the V kernel is based on diskless workstations where all file access is to remote file servers. The system is implemented on a collection of MC68000 based SUN [Bechtolsheim 82] workstations interconnected by a 3 Mb. Ethernet. It takes a controversial approach in building its file access protocol directly on its ipc. facility. A more conventional approach is to design problem oriented protocols for file access (e.g., see LOCUS's file access protocol in Popek [Popek 81]). V, though, does implement its remote operations directly in the kernel rather than through a process-level network server. This avoids unnecessary copying of messages between user and kernel address spaces. What makes this distributed operating system worth examining is that its ipc. primitives are those of Thoth's with suitable extensions.



V Architecture

V , like Thoth, has separate facilities for large and small message transfers. V allows large messages to be transferred directly between user address spaces without any kernel buffering. In V, a client issues a send to a server process which

then receives the message with a **receive**. The server is then allowed to issue one or **moveto** or **movefrom** primitives, which allow large segments to be transferred directly between user address spaces. Finally, a **reply** is issued by the server which ends the message transaction and unblocks the client.

However, these message primitives might seem to be ill suited for a network environment. For example, it would seem that for sequential file access some kind of streaming protocol would be a more efficient use of the network rather than the synchronous request-response communication implied by Thoth's primitives. The file servers reduce disk latency by implementing write-behind and read-ahead which is effective during sequential file access. Similarly, to reduce network latency it would make sense to stream the file pages transferred to the workstation and have the local kernel effectively create a cache of recently read file pages. Since memory is less of an issue with single user workstations, kernel message buffering could conceivably have been designed to implement a cache. These issues are important and forced a performance analysis of this protocol where it is shown that V's approach is just as effective as a more complicated protocol [Cheriton 83]. I will attempt in the second part of this thesis to do a more elaborate performance analysis of V similar to an analysis that was done for LOCUS [Goldberg 83]. Also, this streaming protocol would be non-trivial to implement , create file consistency problems and would be against V's philosophy of simple synchronous primitives.

There are several issues that are raised by this model which must be examined in a distributed environment. How does V guarantee reliable message transmission? How are processes on remote processors identified? What are the file access protocols? The following sections will attempt to answer these questions.

5.1. Remote Message Transmission

V builds a reliable message transmission directly upon an unreliable datagram service. The V kernel collapses the datagram service and the transport service into one. This is possible because a V **reply** provides both an acknowledgement (datagram service) of successful transmission and the actual reply message (transport service). The reasoning for this type of approach is presented in Saltzer's "end-to-end" argument [Saltzer 81] which states that lower level error handling is redone at higher levels in a computer network. Thus, it makes sense to drop some

of the lower level error handling (if errors are rare and can be detected at higher levels) and keep the higher level protocols. Saltzer suggests that remote applications can only be implemented with the help of the applications at either end. This is precisely the implication of using **send-receive-reply**: the server issuing a reply indicates to the client process the success or failure of his transaction. In V, there is no underlying acknowledgement based message system, since this functionality is provided by these primitives. Saltzer argues that the only reason for such an underlying mechanism is for efficiency (i.e., in avoiding unnecessary retransmission of messages).

movefrom and **moveto** follow the same request-response behavior of the **send-receive-reply** primitives. In the case of **moveto**, where a segment is transmitted as maximally sized Ethernet packets, a single acknowledgement packet is sent when the last packet is received by the client. **movefrom** acts similarly, except that the last packet acts as an acknowledgement. This request-response protocol is effective in local area networks where error rates are low and thus full retransmission of packets only slightly degrades performance. Cheriton [Cheriton 84] ,however, describes a more effective protocol which allows for selective retransmission of packets. This protocol does *not* require kernel buffering since it is known where each packet will reside in the since it is known where each packet will reside in the address space of the intended process.

However, V does not extend Saltzer's argument to build a reliable file access protocol directly upon an unreliable datagram service. This approach is seen in LOCUS, a distributed UNIX operating system based on an Ethernet connection, where a problem oriented file access protocol is implemented. In LOCUS, a remote file read would consist of a file request message to the file site which is acknowledged by the delivery of the actual file pages. LOCUS also implements lightweight server processes to run the file access protocol which execute inside the kernel and avoid the processor overhead of scheduling a user level process. In V, however, this transaction would require a **send** to be executed by the client, a **receive** to be executed by the server, a **moveto** to be executed by the server, and finally a **reply** to be executed by the server. In Cheriton [Cheriton 83], this approach is analyzed and shown to be as effective as LOCUS's problem oriented protocol.

When a **send** is issued to a remote process, the kernel calls a routine which performs the non-local send. This routine transmits the send message to the destination processor. The kernel on which the receiving process resides holds the message in a special buffer for a period of time. When the receiving process replies to the **send**, the kernel will also save the reply message in the same special buffer for a period of time. If the sender does not receive a reply within a pre-determined time period the originating kernel will reissue the **send** The receiving kernel will disregard any retransmitted messages by comparing a sequence number and the pid of the client process with that held in the special buffer for the server process. If the message is a retransmission, the kernel discards the message and either retransmits the reply message or else transmits a reply pending message if no reply has been generated by the server yet. The sending kernel will discontinue issuing a send message after a pre-determined number of retransmissions without receiving a reply message or a reply pending message. Of course, if the process does not exist in the remote processor, a negative acknowledgment is sent back and the **send** automatically fails.

Based on the above procedure, we see that reliable message transmission is build directly on an unreliable datagram service.

5.2. Addressing in V

V, like Thoth, addresses a process through a pid. V, though, must insure that pid's are globally unique. The V kernel uses a 32 bit pid in which the higher order 16 bits serve as a logical host identifier while the lower order 16 bits are used as a locally unique identifier. By using an explicit host identifier in the pid it becomes trivial to determine whether a message is destined for a remote processor.

V assumes that there will exist special server processes which can be identified by a logical process identifier. These server processes (e.g., file server, name server, etc.) can be identified with a **getpid** primitive where the user provides a logical pid and the real pid is returned. There is a corresponding **setpid** primitive which binds a logical pid with a pid. In both of these primitives one is allowed to specify a local or remote scope for the logical pid.. In this way, a user can create private servers whose existence is not know globally.

5.2.1. Group Addressing

Cheriton [Cheriton 84] added to V a set of primitives to address a group of processes. That is, the ability of one process to communicate with many. This functionality is naturally provided for by the Ethernet hardware. Primitives are available to allow a process to create a group (which is identified by a group identifier) and for a process to join or leave the group. A **send** to the group identifier means that the message is delivered to all the processes in the group. The first process to issue a **reply** unblocks the sender. In addition a primitive is provided to return any other reply messages which are sent by the other group processes. These primitives would be useful for a process which must *query* a group of servers to locate a specific one or for a process to deliver a new result to many processes in some kind of distributed computation. These primitives might make unnecessary the use of a name server. It would also seem that these primitives supercede the **getpid** and **setpid** primitives.

5.3. File Access Protocols

V's file access protocol is based on an earlier operating system Verex [Cheriton 81]. In V, the **send** message would contain a file descriptor, byte count and an address where the data will be read from (written to). The server after receiving this control information either reads the blocks off a disk and issues a **moveto** or else issues a **movefrom** and writes the appropriate blocks. The reply message is then sent to confirm the amount of data read (written). Notice how the combination of **send-receive-reply** and the large segment transfer primitives are simulating a DMA mechanism. That is, first control information is sent (the control registers are set) and then the transfer is performed. However, even the the designers of V felt that a problem oriented protocol would be appropriate in the case of page level file access. Towards this end, V introduced a **receivewithsegment** and **replywithsegment** primitive. The way this works is that if only a page is being written (read) then the message being sent to the remote processor (the reply message from the remote processor) contains the required page. The **send** message allows for a special kernel format which specifies where in the user's address space the page is to be written to (read from). These two primitives are optional and are transparent to processes simply using a **send**.

5.4. Procedural Interface vs. Message Interface

For many applications, it is preferable to hide the message interface to the V kernel. V, in these instances, provides *stub routines*, that format, send, receive and interpret messages that would be required in the servicing of a request. A *stub* of the procedure is available in the address space of the caller to allow for the parameters which it is called with to be communicated to the actual server process. The remainder of the procedure exists as a separate process on this remote processor. The caller of the routine is entirely unaware of the message processing involved. For example, a process which needs to delay itself for a period of time can call a procedure which in turn sends a message to the clock server. After the required delay period, the clock server sends back a reply message and unblocks the client process which returns from the procedure call with the service accomplished. It is possible, using V's primitives, to provide an efficient procedural interface. The 32 byte size message used in the **send-receive-reply** exchange is adequate to hold an average parameter list (less than 8). A call-by-reference parameter can be implemented by specifying the address where the data resides in the short message and having the receiver access it with one of the remote segment primitives. Finally, the reply message further supports the procedural interface. This is an argument for building a procedural interface and one might conclude that V could have implemented a remote procedure call mechanism directly or RPC [Nelson 81]. This is a logical conclusion if the model of communicating processes is strictly procedural. However, V assumes that its power is not only in its providing a procedural client interface, but also in its efficient message interface.

I have spoken of how an advantageous use of V's primitives is in providing highly concurrent servers, known as administrators. The procedural interface merely requires a server to be a proprietor. The designers of V felt that the ability to write highly concurrent processes directly with the message interface outweighs any advantage the might be available with RPC. For those cases where a procedural interface is preferred, V's implementation of it is efficient. Cheriton questions, in general, the performance of RPC where programs are distributed among different processors purely on procedure call boundaries. V's ipc. is really designed for the writer of distributed programs.

5.5. The Primitives

The following primitives are the ones from V which I would like to use to extend UNIX for use as a distributed operating system:

pid=send(message,pid):	Send a fixed size message specified by *message* to the process specified by *pid*. Block until the receiver has received the message and has sent back a reply message. The reply message overwrites the message area. Return the process identifier (or pid) of the process which has replied.
pid=receive(message):	Block the process if there are no messages waiting. Return the message into the specified buffer. Messages are queued in FCFS order. Return the pid of the sender.
receivid(message,pid):	This is the receive specific form of **receive**. Block until a message from the process specified by *pid* is received.
reply(message,pid):	Send a reply message to the process specified by *pid* provided that this process is blocked on a **send** to the invoking process.
setid(logicalid,pid):	Bind the *logicalid* to the process id. specified by *pid*. This should be a restricted system call.
pid=getid(logicalid):	Return the pid associated with this *logicalid*.
movefrom(spid,dest,src,count):	Copy *count* bytes from the segment starting at *src* in the address space of *spid* to the segment starting at *dest* in the address space of the invoking process. This assumes that the *spid* process is blocked on a **send** to the invoking process.
moveto(rpid,dest,src,count):	Copy *count* bytes to the segment starting at *dest* in the address space of *rpid* from the segment starting at *src* in the address space of the invoking process. This assumes that the *rpid* process is blocked on a **send** to the invoking process.

I am excluding the separate page level transfer primitives to avoid an unnecessary complication of the message primitive set. Cheriton [Cheriton 83] has found that **send-receive-movefrom-reply** sequence performance differs by approximately 3 milliseconds from the more efficient **send-receive-replywithsegment** sequence.

5.6. Conclusions about V

The V kernel is a small communications oriented kernel. Unlike most distributed systems (but like Thoth) it allows for multiple processes sharing an address space. It is intended primarily for process control or graphics applications. Its use of diskless workstations and simple file access protocol based on Thoth's ipc. primitives is appealing and *will* be the basis for our distributed UNIX design. That

is, we are assuming an architecture of single user diskless UNIX workstations which are connected by Ethernet to each other and remote file server stations.

The implications of these primitives for our proposed distributed UNIX presents two issues. The first is the adequacy of the message interface provided by these primitives for UNIX. In general, these primitives would provide a globally unprotected memory, since a client process allows a server full access to its data space. The UNIX environment is general-purpose and requires a better mechanism for protection of the user's memory space. Another problem with these primitives is the assumption that accidental message transmission is rare and when it does occur it is the responsibility of the receiver to filter out bad messages. Again, this should be handled more effectively in a general user environment. Finally, there is the question of whether UNIX requires a transaction level interface rather than the "raw" message interface.

The second issue is the consequences of having the UNIX file system as a remote file server. The file system is central in providing the essential UNIX services It is important to show how these services can exist in the proposed distributed environment.

These issues will be addressed in a later section. However, let us first examine two other distributed UNIX operating systems which have been reported. From these operating systems a practical solution for our distributed UNIX will be motivated.

6. Two Distributed UNIX Operating Systems: MIMAS and S/F-UNIX

There have been attempts at extending UNIX for a distributed local area network environment. MIMAS [Blair 83] presents a set of transaction level primitives which provide a remote procedure call mechanism. This is accomplished by building these transaction level primitives upon an underlying datagram service which is itself based upon UNIX I/O system calls. This is certainly one way to provide UNIX ipc. which avoids the problems of a separate ipc. system. Other distributed UNIX's use a similar technique [Haverty 78]. Our purpose in examining it is to show that our proposed UNIX is capable of the same functionality as an existing distributed UNIX system *and* provides an efficient message interface in

which to write highly concurrent programs.

S/F-UNIX [Luderer 81] presents a simple approach to accommodate a remote file server for the UNIX environment. Its approach of translating all file requests into a corresponding message transaction will be the basis for designing our distributed file system.

6.1. MIMAS

The designers of MIMAS decided on building an ipc. capability directly on UNIX I/O rather than providing special ipc. primitives. Their major consideration was to make as little change to the UNIX kernel as possible. As was mentioned previously, UNIX device I/O is closely connected to the file system in that a character I/O device is treated as a special file. In MIMAS, the ipc. is built on the UNIX I/O system calls (**open,read,write,close,seek,ioctl**). These system calls are the basis of a datagram service on which a more reliable transaction service is built. At the lowest level of communication, a process is identified by a port number and a station address (the network location of the processor). The UNIX file system is able to associate a port number with a file name within the UNIX *i-node* structure. A process wishing to initiate communication must have a port on which to send and receive messages. A process, then, must issue an **open** to a special file which is the DMA driver for the network. A **write** call initiates the transmission of network packet and returns the success or failure of the transmission. A **read** call is blocking and waits until a message arrives on the specified port. Another UNIX system call, **ioctl**, allows a port to be interrogated for status information. From the **read** and **write** system calls MIMAS builds a non-blocking **send** and a blocking **receive** datagram primitives. From the **open**, **close**, and **ioctl** system calls, primitives to create, destroy, and interrogate a port are provided. Therefore, there is a direct mapping between the datagram primitives and the UNIX I/O system calls.

The next layer that is built is designed to provide a remote procedure call mechanism. MIMAS is based on the client-server model whereby users communicate with remote processes as if they were procedures. A client issues a **remote_call** which specifies the server to be called (a station-port number pairing), a list of parameters to be sent to the server, and the buffer in which results from the server may be returned. The server receives these parameters by a **receive_request**

call. In this call, one also specifies a port number parameter from which messages will be received and the port number of the client which sent the message is returned in another parameter. The server eventually issues a **send_reply** to the client's port number and returns any results. Each of these transaction primitives is built directly from the datagram primitives specified above. By basing the datagram primitives on UNIX I/O system calls and by using a separate processor to run the DMA driver, MIMAS is a UNIX extension where very little UNIX kernel code is changed.

MIMAS avoids designing separate ipc. primitives and is forced to build datagram primitives directly on UNIX I/O. However, with our proposed distributed UNIX, ipc. is an essential kernel service. We gain all the advantages of the V kernel while avoiding the inefficiencies of relying on the UNIX file system. MIMAS makes the assumption that most distributed UNIX programs can be divided on procedural boundaries. While this might be true for most applications written by users, MIMAS does not support an efficient message interface for more general process communication. As argued in a previous section, the advantages of V's primitives are in its power to create highly concurrent distributed programs. In addition, it would be relatively easy to create a transaction service for our distributed UNIX which is very similar to MIMAS's. The major advantage of **send-receive-reply** is precisely that it provides both an efficient message interface and a basic procedural interface.

6.2. S/F-UNIX

We have seen an approach which attempts to build a new transaction level service on top of the UNIX I/O system calls. A much more conventional approach is seen in S/F-UNIX. In this distributed architecture, UNIX workstations (which includes small disks to hold a local directory) are connected by a high bandwidth virtual circuit switch to file servers and other peripherals. This architecture allows terminals to be connected directly to the data switch and not associated with any processor. The workstations run a UNIX operating system known as S-UNIX which allows message based access to remote files. The file servers run a specialized UNIX known as F-UNIX. This system, though, does not provide any specialized ipc. primitives. The system does not allow workstation to workstation

communication except through UNIX *fifo*'s. This, of course, implies that remote ipc. has the overhead of the file system. Although MIMAS does lack many of the ipc. facilities usually associated with a distributed operating system, this is in fact a distributed operating system! There is a globally shared file system. There is also the concept of a pool of workstations and other processors which allows a locally created process to execute remotely. What is relevant about this operating system, though, is the way remote file service is provided by the local S-UNIX kernel. The S-UNIX kernel translates all remote file requests as an appropriate message to a file server running on the F-UNIX machine.

S-UNIX, unlike our proposed UNIX workstations, does maintain a small local file system. S-UNIX can refer to a file or directory on the remote file server by establishing an *i-node* of type *remote*. When the local S-UNIX kernel is translating a path name and crosses a *remote inode* then the remaining path name is sent to the file server. After a remote file is opened, the local S-UNIX kernel maintains the pid of the file server and a unique number assigned by the file server (actually the *i-number*). It also maintains the *i-number* of the current and root directories. All other information about a file is maintained by the remote file server. When a process needs to perform an I/O operation on a remote file it sends a message to the remote file server along with the appropriate i-node. When a path name of a file to be opened is sent to the file server, the *i-number* of the current or root directory is also provided depending on whether an absolute or relative path name is sent. S-UNIX does not maintain a cache of remote file pages or *i-nodes*. This approach avoids any elaborate message-based locking mechanism.

The F-UNIX operating system was designed specifically to run the file server process as a kernel process. F-UNIX does not provide a full UNIX process environment. However, the designers believe that providing a UNIX environment would be an advantage. They point out that this would allow the UNIX file check and repair programs (e.g., *fsck*) to be run on the F-UNIX machine as a process. F-UNIX requires operator intervention to manually repair a damaged file system in the current implementation. This raises an important design questions for file server machines. Should there be a specialized operating system designed to support just the file server processes (i.e., a real-time system) or should the operating system be more like the user's operating system? The suggestion is that

they should be more like the user's operating system so that they will be capable of running user programs. This issue will be discussed in more detail when a UNIX file server implementation is presented.

The assumption with our distributed UNIX is that individual workstations are diskless and use V's ipc. primitives for all remote communication. Since there is no local file system, we are forced to build UNIX I/O system calls themselves on V's ipc. primitives. There is a direct mapping between UNIX I/O and V's primitives. **open**, **close**, **seek** and **ioctl** would imply a **send-receive-reply**. **read** and **write** would imply a **send-receive-movefrom/moveto-reply**. Many of the remaining I/O system calls would involve the simple **send-receive-reply** transaction (e.g., **link**, **stat**, **access**, etc.). Thus, the UNIX I/O system calls would be provided as part of a run-time library in the address space of the UNIX process. The formatting, sending and receiving of messages involved in the protocol will be invisible to the program. This, of course, is the approach that V takes for its file access.

7. An Approach

Unlike MIMAS, our distributed UNIX provides a powerful set of message passing primitives which would allow the general the user the capability of writing highly concurrent programs. Higher level transaction primitives could be designed from V's primitives as stub routines. This would provide a procedural interface for the user who does not require highly concurrent programs.

There are two issues which must be resolved for our distributed UNIX. The first is that as a message interface, V's primitives do not provide an adequate level of protection consistent with the UNIX environment. The second issue is how do we preserve the essential UNIX services (hierarchical file system, device independence, and asynchronous processes) in the proposed distributed environment. The following sections will present an approach to these issues.

7.1. Protection

Some consideration must be given to unrestricted ipc. traffic in a distributed environment. Although it is possible to require a process to verify each message received, a better approach is to initially insure that only certain messages will be

received. UNIX has a simple but effective scheme of file protection that could be extended to the ipc. problem. UNIX assumes that a user belongs to a group of related users. Each user has associated with him a user-id and a group-id. A user can extend privileges (read,write,execute) to users with the same group-id to all users or to no other user. We can also provide this same protection structure to message traffic. To avoid adding a new primitive, we can assign a pid to correspond to the UNIX kernel and allow a user process to communicate its protection information to the kernel. It is generally a useful technique in message-based systems to be able to communicate directly with the kernel as if it were a process. The V kernel allows such functionality [Cheriton 84] and we should borrow this idea for our UNIX system. A process will specify who can communicate with it by setting a permission bit for user, group, or all groups. A process that wishes to communicate with a given process must have a permission to do so or else the kernel will reject the message with an appropriate error message. This implies that appended to each message must be the user-id and group-id of the process sending this information. It is the responsibility of the kernel to supply this. This information is needed by the file server to determine access permissions, so it would have been required anyway to implement the UNIX file system.

After a server has issued a **receive**, this process has the ability to read or write anywhere in the client's data space. In UNIX, this means that both the stack and data segments can be accessed. We have seen that the most common use of large message transfer is in the file server protocol. The client's address space is totally accessible to the server. In a general user environment this could be a problem. The way out of this is to adopt the V convention which allows a process to specify which segment in the data space would be accessible to the server [Cheriton 83]. By segment, I do not mean one of the UNIX *system* segments, but rather what the user would consider to be a continuous region in his data space. It would be the user's responsibility to append this information along with the send message as part of a special message format which the kernel can interpret. For example, the client process would specify a starting address in the data space, a length, and access permissions. Then, when it comes time for the server to issue a **movefrom** or **moveto** primitive the local kernel can make sure that this process has permission to read or write the segment.

7.2. Terminal Communication and the Splice primitive

An interesting problem occurs in UNIX because terminal I/O is treated as a special file. In our implementation, this means that a message must first be made to the file server, which is a remote processor. However, all read and write requests must be sent to the user's terminal which is on the same processor. This implies that the file server must hand off a send message to another processor. This operation can be formalized with the **splice** primitive. It definition is as follows:

splice(spid,rpid,message)
Allow the process specified by *rpid* to reply to the process specified by *spid*. This assumes that *spid* is blocked on a **send** to the invoking process. The process specified by *spid* will now be blocked on a **send** to the *rpid* process. The *rpid* process will receive the specified message when it executes the **receive** primitive.

The splice primitive is essentially the Thoth **forward** primitive. Although Gentleman feels its use is not justified in general in distributed systems, it is critical in this UNIX extension and can be found in the V kernel. Also, I am clarifying the semantics of this primitive by requiring that **send** will return the pid of the spliced process. We require the **splice** primitive to be non-blocking. Its semantics imply that the message will be sent to *rpid* and that this process will issue a **receive-reply**. When the reply message is returned, then the **send** returns the pid of the spliced process (*rpid*). Therefore, when the remote file server receives the open request for a terminal device, the file server splices the message to the terminal server process on the original processor. After the terminal server replies to this request, all subsequent requests are sent directly to the terminal server, since it is this pid that has been returned from the original **send**. When the terminal server has received a close message, it will perform a close on the terminal and then perform a *splice* back to the original file server process. The file server then performs some file system book keeping and issues the final **reply**.

The splice primitive helps to preserve device independence by allowing the file server to transfer control to the server for the appropriate device.

The splice primitive is also useful in multi-process structuring. That is, it would be not an uncommon situation for the administrator to create a process to handle a client's request and then splice the client to the newly created process. This would increase concurrency by having every client associated with a special

created process (at the expense of stack and data space).

7.3. File System Considerations

The problem with having a remote file server is deciding how information should be split between the local and the remote operating systems. In the S/F-UNIX operating system, S-UNIX translates all remote file requests into appropriate messages for the file server running on the F-UNIX system. As we have seen, S-UNIX maintains very little file information locally.

It is essentially this approach that I would like to employ for our distributed UNIX system. Our local UNIX kernels will not maintain any file system information locally. Each *process*, though, will maintain the pid of a file server and a *file descriptor* number which can be used on all successive operations. The file server must now maintain the mapping between the *file descriptor* and the *i-number*. It must also maintain the current and root directories for each process. All UNIX file system calls will then be translated to an appropriate message and sent to the file server.

An obvious optimization is to translate common file operations (copy, move, compare, etc.) into appropriate messages which are then run by the remote server. It should be remembered that UNIX executes many file operations as applications programs rather than providing them as a kernel service. So, it would certainly be more efficient to run these programs on the remote file processor rather than waiting to load them to the local processor and executing them there.

7.3.1. File Server Implementation

A server implementation of the UNIX file system based on an administrator offers an elegant design possibility. Gentleman [Gentleman 80] speaks of creating two administrators for the UNIX file system, a path name administrator and a backing store administrator. The backing store administrator maintains the flat file system which treats the disk as a linear array of blocks. The path name administrator uses the linear address space provided by the backing store administrator to create a directory hierarchy.

The path name administrator will respond to an open request, for example, by replying with the pid of the backing store administrator and a token (i.e., a UNIX *file descriptor*) to identify the disk address space of the process. The path name administrator, in order to translate a path string, will communicate with the backing store administrator through one or more courier processes. The courier will be communicating the *i-node* for each of the directories in the path name. The backing store administrator will take this *i-node* and presumably be returning the disk blocks where the current directory entry in the path string is found. This process is repeated until the last directory entry where the file resides is returned. Thus, the path name administrator provides the tree structure. After the reply to an open is returned, the client process can communicate directly with the backing store administrator by providing it with a *file descriptor* which will map directly into an *i-node*. The backing store administrator will translate a logical block number into an actual physical address , or better yet, have a worker process do this. There can then be a worker process for each disk drive which will perform the actual read/write/seek and then notify the backing store administrator when this action is completed.

This solution provides adequate concurrency by allowing the path name administrator to service requests destined for several different disks while the backing store administrator performs the actual I/O. This is an interesting solution to the problem but probably not practical in the UNIX environment. UNIX is a time-sharing operating system and some of these processes would require real-time scheduling (the backing store administrator and its worker processes). While it might be possible to design a small real-time operating system just for the file server machine, it is a better policy to have a UNIX-like operating system running there. As mentioned before, there would be a performance and operational advantage in providing UNIX functionality to the file server machine. For example, I/O intensive operations, like the copying of files, could be run as a UNIX program on the file server machine. Presumably, the file server will interpret messages requiring this type of activity by having a worker process **fork** and **exec** one of these programs.

If we are not providing a real-time operating system to support the server, this implies that the kernel must provide some kind of basic I/O service for the file

server process. Cheriton [Cheriton 84] discusses a common I/O protocol for V that would be implemented by the kernel and would allow uniform access to all devices. He notes that this would be especially useful idea for an operating system like UNIX which treats all I/O as file-like devices. He suggests a small set of primitives for the protocol which is based on access to a uniform I/O object, or UIO [Cheriton 84]. A UIO is essentially a UNIX *i-node*. He provides primitives to create (or open) a UIO, destroy a UIO, query a UIO, update UIO information, and read or write to the file represented by the UIO. A segment is read or written by providing a UIO and by specifying the logical file block number of the start of the data transmission. The UIO, like the UNIX *i-node*, is maintained as a kernel data structure.

The suggestion that is being made is to have the UNIX kernel provide a similar set of primitives which would act on a UNIX *i-node*. This implies that the backing-store administrator, mentioned above, would be provided as a kernel service. This avoids the problems of providing real-time process control for our file server process. The file server process, then, would essentially be a path name administrator. That is, it will access a UNIX *i-node* and then perform some action on it based on the message it is currently processing. A few of the UNIX I/O calls involve direct access to an *i-node* (e.g., **fstat**, **fcntl**, and **dup**.) The path name administrator can respond to these requests quickly. Most of the I/O system calls require a path name to be translated into an *i-node* (e.g., **stat**, **link**, **unlink**,**chmod**,**chown**,**access**, etc.) and would involve considerably more processing. However, the file server should be capable of maintaining a software cache of recent disk block requests, just as in the UNIX file system, and provide a more reasonable response time.

By having the kernel for the file server provide essential services we keep its size small. The file server process should be able to take advantage of the extra memory which comes with being on a separate machine for cacheing of disk blocks. Another advantage to having a path name administrator process is the concurrency possible if we allow several such processes each to administrate their own logical file system. A single administrator can also accomodate any inherent concurrency by creating worker processes, as needed, to perform some particular function that is required by an individual request. In general, a single administator is used to guard a resource (in this case a logical file system) and worker processes (which can be

dynamically created by the administrator) perform logically concurrent tasks.

7.3.2. UNIX Fork

A UNIX **fork** requires that a child process shares all open files with its parent. This implies that the file server must be made aware that a child process has been created. Therefore, the local kernel must send a **fork** message to the remote file server. On receipt of this message (which should contain the pid's of the parent and child process), the file server will create a duplicate set of *file descriptors* for the child process which will map into the same open files as the parent process. Therefore, when the child process attempts to perform an I/O operation on a file, the file server will accept the child's *file descriptor* as valid. If the parent process has any devices open, the file server will also need to send **fork** messages to the appropriate device servers. The device server must also be made aware of the new child process to allow the parent's *file descriptors* to be duplicated for the child process.

The above procedure requires that the kernel be capable sending and receiving messages. I have mentioned previously that in message-based operating systems this is useful property for the kernel to possess. We are fortunate that the UNIX kernel easily allows for this property since it supports the notion of kernel processes.

7.4. Conclusions about a Distributed UNIX

From the above discussion, a distributed UNIX is being suggested with essentially the same ipc. set as the V kernel. These primitives will be used as the basis for the file protocol to the remote file server. They will also be available to the general user to write concurrent programs. Although it would be possible to build a remote procedure call mechanism, this decision should be made on the basis of the particular application. It is proposed that some kind of message protection could be provided by limiting which processes can send to each other on the basis of the UNIX user-id and group-id names. In addition, to protect the client's address space from an inadvertent read or write, the client will be able to specify in his send message which segment is available to the server. The local kernel will not maintain any file information except perhaps the logical process id. to real process id.

mapping for the remote file server. The file server will not run on a specialized operating system but rather a UNIX operating system. However, this kernel will provide system calls which allow the file server direct access to an *inode*. The file server will act as a path name administrator leaving the actual I/O service to the kernel. Finally, a way in which the essential UNIX characteristics will be preserved has been described.

8. UNIX Implementation of Send, Receive, and Reply

The UNIX kernel can be divided into 3 functional areas: process control, I/O system, and the file system. In implementing these basic ipc. primitives most of the code can be restricted to the process control section of the kernel. Although a full implementation of all the primitives would require additional changes to the I/O system. The code that was implemented was run under Berkeley 2.9BSD UNIX on a PDP-11/23.

The UNIX kernel proves to be very adaptable to changes primarily because most of the code is written in C. In adding message passing primitives to the kernel, one must be aware of how the kernel implements process synchronization. Processes wait for events. An event is represented by an arbitrary integer. Typically, an event is the address of some kernel data structure that is associated with the event. A process gives up the processor by calling the *sleep* routine and specifying an event to be blocked on. A process can unblock other processes which are waiting on a event through a call to the *wakeup* routine. A *wakeup* unblocks all processes waiting on that event. Typically, a process which is unblocked must check to see that the reason it blocked itself had gone away; if it has not, it blocks itself again. The *sleep-wakeup* calls imply that system processes (processes executing within the kernel) are like co-routines. A system process can assume that no other process is executing concurrently with it. This is not quite true, though. The kernel for the most part runs interruptibly. This can cause unpleasant race conditions when the interrupt process attempts to change important kernel data structures and issue a *wakeup* call. This problem can be avoided by carefully inhibiting interrupts at certain critical sections within the kernel. For a more complete discussion of this, see Lions [Lions 77].

The major changes to the UNIX kernel data structures were adding a message buffer to the process table and a new data structure on which processes may queue themselves when they make send requests. A time out mechanism was added so that the server must remove the client's message within a given amount of time or else control is returned to the client and an error code is set indicating a timeout. The implementation of these primitives can be found in Appendix A.

9. Design Conclusions

In the first part of this thesis, an approach to message passing was presented to be used as the basis for designing a distributed UNIX operating system. Their use in Thoth and V suggested that they would be appropriate for the UNIX environment. The goal, in general, was to provide a simple set of primitives which was consistent with the overall UNIX philosophy of avoiding unnecessary complexity. This would allow UNIX to support server processes. This provides the potential for moving basic operating system services (e.g., the file system) out of the kernel and onto other processors as separate processes. It also allows for concurrent applications in general to be created by the user. The file system is the most logical operating system service to be implemented as a server and therefore create a distributed UNIX environment. A solution to designing a remote file server which avoided complicated concurrency control was presented. However, the performance implications of having remote file service in UNIX have not been explored. In the second part of this thesis we will consider a performance analysis of this issue.

Performance Analysis

10. Introduction to Performance Analysis

In the first part of this thesis a set of efficient message passing primitives has been introduced. We have suggested using these primitives to extend the UNIX operating system for a distributed environment. An important part in designing a new extension to an operating system is a performance analysis of these features. In general, we are interested in such things as the "work load" that can be supported and determining where bottlenecks will arise.

We are somewhat fortunate in that the V operating system is well documented and we do have some empirical evidence of its network performance. The evidence is used to support the claim that remote communication is competitive with local communication and that concurrency is gained when the local processor is offloaded. Cheriton [Cheriton 84] was also interested in showing that sequential file access is efficient, even though a streaming protocol is not implemented. This analysis is good, as far as it goes. However, what is not discussed is the average behavior of these primitives in a multi-user environment. It is this aspect which will be analyzed for our UNIX extension using techniques from queueing network theory. In particular, we will be interested in analyzing the transfer of large segments between the file servers and the local processors. This is especially the case in UNIX where most of the file activity involves the loading of programs off of the disk.

Although we are interested in the network performance of these primitives, another consideration is how these primitives perform locally. This is an important consideration and **send-receive-reply** would seem to be fast primitives. However, Cheriton [Cheriton 84b] considers the use of exploiting the registers of the Motorola 68000 as a way to further improve performance. This is accomplished by having communicating processes share a set of registers and pass the send and reply messages through these registers.

Let us briefly review the network performance results of the V kernel.

11. V kernel performance

File I/O is the dominant use of V's ipc. primitives. The time for a single file page of 512 bytes to be transmitted is the time for the sequence **send-receivewithsegment-reply** to occur. A page write operation occurs in the reverse order and we can assume it takes the same amount of time. A less optimal sequence of **send-receive-moveto-reply** is possible but let us consider just the specialized primitives. Page level file access using these special primitives indicates what file performance would be like if the page was already in memory.

Page Level File Access				
	Elapsed Time		Processor Time	
Operation	Local	Remote	Client	Server
page read	1.31	5.56	2.5	3.38
512 byte page (time in milliseconds)				

The difference between a local and remote page read, 4.2 milliseconds, is small compared to the time for the actual file server operation. We can estimate disk latency to be about 20 milliseconds and file server processor time to be about 2.5 milliseconds (based on the LOCUS file server.) Thus, total local file access time is 23.1 milliseconds and remote time is 28.1. This makes the remote file operation only 18% more expensive. Remote file access also offloads the local processor. If the time to perform the file operation is greater than the difference between the processor time for local page access and remote page access (2.5 - 1.3= 1.2) then concurrency is gained. That is, the local processor is freed for that amount of time to run another process. Finally, if the remote file processor, by using larger disks and more memory for disk caching can lower the time for a page level file operation by 4.2 milliseconds, then there is an advantage to using a remote file processor.

The preceding discussion was meant to show that there was no overwhelming disadvantage to using V's primitives. However, the predominate type of file activity,

sequential file access, would appear to be a problem for the non-streaming synchronous V file access protocol. That is, a streaming protocol attempts to reduce network latency by transmitting pages before they are required, just as disk read-ahead reduces disk latency. However, when network latency is small; as it is in Ethernet, and the time for file access is dominated by the the disk access time, then a streaming protocol can be dispensed with. Cheriton [Cheriton 84] modified the experiment which gave the file page read times (above table) by interposing a delay in the file server process to simulate disk latency. This delay occurs after a reply message is sent and before the receipt of the next message. This approximates the behavior of a program sequentially reading as fast as possible from a file server which is reading ahead.

Sequential Page Level Access	
Disk Latency	Elapsed Time per Page Read
10	12.02
15	17.13
20	22.22
512 byte page (time in milliseconds)	

Since disk latency is the lower bound on remote file I/O, we see that the absence of a network read-ahead (or streaming) protocol in V's ipc. introduces a performance penalty of at most 15%. Network read-ahead would be effective in the case where an application is reading blocks slower than the file server can deliver them. If it were reading them faster than their availability, the streaming protocols' behavior would be the same as V's. In the former case, the improvement is small. If the disk latency time is 20 milliseconds and the application requests pages every 20 milliseconds, then with the streaming protocol the page would be available locally and would require only 1.3 milliseconds to get. With V's I/O protocol this would require 5.6 milliseconds. However, since read request are being sent every 20 milliseconds the savings in total time between request is merely 20%. This is too small a savings to justify the increase in the complexity of V's protocol.

In V, **movefrom** and **moveto** primitives can be used to transfer large segments efficiently between processes. In the UNIX operating system, practically all commands entered from the terminal force the UNIX kernel to load a program from the disk and create a process to run the program. This implies that in our distributed UNIX based on V's ipc. primitives, most of the file activity is the transfer of large segments between the local processors and the remote file servers. The rest of the thesis will attempt to analyze this behavior in a multi-user environment based on modeling this system as a queueing network.

12. Review of Queueing Network Theory

Networks of queues are important models for multi-programmed computer systems, distributed computer systems, and communications networks. A queuing network is a collection of stations (or service centers) where customers demand a service before proceeding to another station to fulfill their total service requirements. CPU's, channels, and disks can very naturally be considered as stations in the computer system. Jobs would then be the customers. In a distributed system, workstations would be the service centers while the messages generated would be the customers.

Early queueing analysis of computer systems used the most basic results of queuing theory. That is, computer systems were considered to be a single queue with an infinite customer population. The classic M/M/1 queuing system was used to completely specify such a system. This describes a single queue with a Poisson arrival process (the time between customer arrivals is exponentially distributed) and the service time required by each customer also has an exponential distribution. The steady state probability solution of finding k customers in the system exists and exhibits the well known product form where λ is the arrival rate and μ is the service rate:

$$P(k) = \left(1-\frac{\lambda}{\mu}\right)\left(\frac{\lambda}{\mu}\right)^k$$

However, for the purpose of modeling computer systems, isolated queues are of very little use. The internal structure of the computer is not well represented and classic queuing theory assumes an infinite population of customers. A computer

model must assume that there are a finite number of users. Fortunately, queueing theory has developed models for networks of queues with a finite population or closed queueing networks. What is remarkable about the steady state solutions of such networks is that they too exhibit the product form solution.

12.1. Burke's and Koenigsberg's Work

Burke [Burke 56] was able to show that the product form solution existed for a series of M/M/1 queues connected to each other. That is, the steady state probability of a system of n queues was the product of the steady state probabilities of n individual M/M/1 queues. This interesting result was a consequence of the fact that the steady state departure process of a M/M/1 queue is Poisson with rate λ if the arrival process is Poisson with rate λ ! Thus, a series of queues connected to each other behave as independent M/M/1 systems since the joint probability function factors into the marginal probabilities. We will see that this result holds for more complicated networks of queues.

Koenigsberg [Koenigsberg 58] introduced a closed system of queues which he called a cyclic queue. It is essentially Burke's series of M/M/1 queues where the output of the last queue is fed into the input of the first, so a cycle is formed. Koenigsberg solved this model assuming a finite customer population. This solution has the product form, although we now no longer have a product of marginal probabilities because of the dependency on the number of customers in the system. The Koenigsberg model is important since it begins to approximate a real computer system.

12.2. Jackson's Work

In 1957, J.R. Jackson [Jackson 57] took these basic results and extended them into a solution of an arbitrarily connected open network of servers with exponential service time distributions. In an open network, customers are allowed to enter any station in the network from outside the network. The assumption is that their arrival is described by some Poisson process. Actually, Jackson presented a solution *technique* and not until later was an efficient computational algorithm developed.

Let us present a somewhat restricted case of Jackson's solution. Assume we have a network of M stations where the ith station consists of a single M/M/1 server. After leaving the ith station a customer proceeds to the jth station with probability θ_{ij}. He will leave the system altogether with probability $1-\sum_{j=1}^{M}\theta_{ij}$. Jackson's solution technique involves solving simpler local balance equations in order solve the global balance equation. This involves first solving a system of linear equations which equates the total rate at which customers are entering a particular server with the rate at which they are leaving:

$$Y_j=\lambda_j + \sum_{i=1}^{M} Y_i\theta_{ij} \text{ , for } j=1,2...M$$

In the above equation, λ_j is the rate of the Poisson process which represents customers entering the jth station from outside the network. Jackson equates the total arrival rate at a station to be the arrival rate of customers from outside the network plus the arrival rate of customers from all internal stations. These linear equations will have a solution. (This requires that the underlying Markov process is irreducible, i.e. every state of the system can be reached from every other state [Kleinrock 75].) Jackson was able to show that the steady state probability for the entire system, $P(s_1,s_2,...,s_M)$ is equal to :

$$p_1(s_1)p_2(s_2)\cdots p_M(s_M)$$

The terms are the probability functions of the isolated queue with arrival rate Y_i and service rate μ_i. Thus, the joint probability function can factor into the marginal probabilities and so the the servers are independent of each other! This is a remarkable result and would seem to be the logical endpoint of some of the above ideas.

Jackson's solution technique involves the principle of local balance. Briefly, a global balance equation is obtained from the differential equation which describes all the ways in which a state of the system can be reached. Intuitively, we are equating the *rate* at which the system departs from a state to the rate at which it enters this state. By using the principle of local balance, we equate a subset of terms on the left side of the equation to those on the right. If we can show that all the subsets of the global equation balances, then the original global balance

equation must be satisfied. For example, Jackson applies the principle of local balance by equating the steady state rate at which customers are leaving all the stations in the network with the rate at which customer are coming into these stations from outside the network plus the steady state rate at which they enter theses stations from other stations in the network:

$$\sum_{j=1}^{M} \mu_j P(s_1,s_2,\ldots,s_M) =$$

$$\sum_{j=1}^{M} \lambda_j P(s_1,\ldots,s_j-1,\ldots,s_M) +$$

$$\sum_{l=1}^{M}\sum_{j=1}^{M} \mu_l \theta_{lj} P(s_1,\ldots,s_l+1,\ldots,s_j-1,\ldots s_M)$$

Jackson then applies the principle of local balance again to obtain *local balance equations*. Local balance equations equate the rate at which the system departs from a given state due to the arrival of a customer to a particular queue to the rate at which the system leaves this state due to the departure of a customer from this queue [Chandy 72]. These local balance equations have the form:

$$\mu_j P(s_1,s_2,\ldots,s_M) = \lambda_j P j + \sum_{l=1}^{M} \mu_l \theta_{lj} P(s_1,\ldots,s_l+1,\ldots,s_j-1,\ldots s_M) \text{ , for } j=1..M$$

By then assuming that $P(s_1,s_2,\ldots,s_M)$ takes a product solution of the form, $(Y_1/\mu_1)^{s_1}(Y_2/\mu_2)^{s_2}\cdots(Y_M/\mu_M)^{s_M}$ (where Y_l is as defined previously), the local balance equation is then shown to balance and the original subset of the global balance equation must also balance. This procedure is then repeated for another subset of the global balance equation and finally the original global equation is then shown to balance. Jackson's results can be shown to be true for a mixed system of M/M/1 queues and M/M/n queues (a station consisting of n parallel servers). This can be summarized below:

$$P(s_1,\cdots,s_M) =$$

$$\prod_{l=1}^{M} p_l(s_l)$$

$$p_i(s_i) = \left\{ c_i(Y_i/\mu_i)^{s_i}/D_i(s_i) \right.$$

$$D_i(s_i) = \begin{cases} s_i! & s_i \le r_i \\ r_i! r_i^{s_i - r_i} & s_i > r_i \end{cases}$$

where r_i is the number of parallel servers at server i

and $c_i = (1 - Y_i/\mu_i)$

Jackson extended the above results [Jackson 63] to include a closed queueing network (first considered by Koenigsberg) as a special case. He was able to create this condition through the mechanism of a "triggered arrival". That is, whenever the total number of customers dropped below a certain predefined value as a result of a customer exiting the system, a new customer is injected. This can then be interpreted as a closed system with a fixed number of customers. Jackson also generalized the notion of parallel servers. In a system with r parallels servers each with a service rate of μ the total service rate can be represented as $\mu \cdot min(r,k)$. Jackson allowed the service rate to be some arbitrary function of the number of customers in the system.

12.3. Gordon and Newell's Work

Working independently, Gordon and Newell [Gordon 67] considered arbitrarily connected networks which are closed. Their results are really a special case of the more general Jackson model [Gordon 67b]. That is, the number of customers in the network are fixed and can neither leave or enter the network. The solution technique is practically the same as Jackson's and involves solving a simpler set of local balance equations by first solving a set of linear equations which equates arrival and departure rates to a particular queue. The steady state solution exhibits the product form solution, i.e. there is a factor for each station in the network. Their results are:

$$P(s_1,\ldots,s_M) = \frac{\prod_{i=1}^{M} X_i^{s_i}/D_i(s_i)}{G(N)}$$

where $X_i = e_i/\mu_i$, μ_i is the mean service rate for the ith server

$$e_i = \sum_{j=1}^{M} e_j \theta_{ji}$$

$$D_i(s_i) = \begin{cases} 1 & s_i = 0 \\ \prod_{l=1}^{s_i} d_l & s_i > 0 \end{cases}$$

where θ_{ji} is the probability transition of a customer from station i to station j;

e_i can be interpreted as the relative throughput at station i;

and d_i is an arbitrary positive function reflecting the service rate multiple when there are i customers present

$$G(N) = \sum_{s \in S} \prod_{i=1}^{M} X_i^{s_i} / D_i(s_i)$$

$$\text{where } S = \left\{ \text{for all } s_i \text{ such that } \sum_{i=1}^{M} s_i = N \text{ and } s_i \geq 0 \right\}$$

In the above, G(N) is the normalization constant chosen to make all the feasible states in this closed system sum to 1. It should be noted the difficulty in obtaining G(N) . It involves enumerating the number of ways N customers can be distributed among M service stations or:

$$\binom{M+N-1}{N}$$

This means that an efficient solution to G(N) is critical to the application of the above results. It should be noted that there are other techniques in solving closed queueing networks which can be competitive with a straight foward enumeration of the Gordon and Newell steady state equation. There is the so-called recursive solution methods which involves solving the equilibrium balance equations (see Chandy [Sauer 81]). The algorithm gets its name because of the form of the equations used. This an effective technique only when the number of customers and stations is small. Another approach to the solution of these balance equations is the iterative method. This method provides an approximate solution to the balance equations which has been shown to converge to the actual equilibrium solution

[Wallace 66]. However, these methods while useful in special cases (where there does not exist a product form solution) are superceded by Buzen's efficient computation of G(N).

12.4. Buzen's Thesis

Buzen [Buzen 71, Buzen 73] presented both a valid model of multiprogramming system and an efficient way to compute the normalization constant. In his thesis he essentially restates Gordon and Newell's results for closed networks. However, he also presents an efficient technique for computing the normalization constant (known as G(N) where N is the number of customers in the network). In addition, he shows how all of the important performance measurements of the system can be calculated in terms of G(N). I would like to briefly review the Buzen technique.

The approach I would like to take to derive the Buzen results is from Shum [Shum 76]. She notes that the solution to the closed network is that of the open network constrained by the number of customers in the system. Kleinrock [Kleinrock 75] shows that :

$$\textit{Open system: } P^{\circ}(s_1,s_2,\ldots,s_M) = F(s_1)\cdots F(s_M)$$

$$\textit{Closed system: } P(s_1,s_2,\ldots,s_M) = \frac{F(s_1)\cdots F(s_M)}{G(N)}$$

$$= \frac{P^{\circ}(s_1,s_2,\ldots,s_M)}{G(N)}$$

Thus, we can evaluate G(N) as:

$$G(N) = Pr\left\{\sum_{i=1}^{M} s_i = N\right\}$$

We have now reduced the problem to evaluating the probability distribution of the sum of m independent variables. (Jackson has shown that the probability distribution of the number of customers at each server is independent of any of the other servers). Therefore, we can use tools of probability: convolutions and generating functions. Since the marginal probabilities of the individual servers are independent of each other (in the corresponding open system model), the

computation of the sum of these random variables becomes more manageable. That is, let $P(H_i=s_i) = F(s_i)$ be the probability distribution of the random variable H_i. Let us assume that H_i is describing a single server station. This is a server whose processing rate is not dependent on the number of customers being serviced and is known as a load independent server. Then, we are interested in the probability distribution of $H = \sum_{i=1}^{M} H_i$ and $G(n) = Pr(H=n)$. This is clearly the convolution of M independent random variables.

We know that if $\Phi_{H_i}(t)$ is the probability generating function of the random variable H_i then:

$$\Phi_H(t)=\Phi_{H_1}(t)\Phi_{H_2}(t)\cdots H_{H_1}(t)$$

And that if g(n,m) is the coefficient of t^n in the polynomial after the mth convolution, then g(N,M) = G(N). Let us look at the probability generating function of the random variable H_i as defined above. Then:

$$\Phi_{H_i}(t) = k_i(1+X_it+X_i^2t^2\cdots)$$

This is so since the probability distribution of an M/M/1 queue is geometrically distributed. Therefore:

$$\Phi_H(t) = \prod_{i=1}^{M} k_i\Phi_{H_i}(t)$$

$$= \prod_{i=1}^{M} \frac{1}{1-X_it}$$

As Shum notes we can ignore the constant factor.

Now, there is a simple recursive method to evaluate g(n,m). Let:

$$h(1) = \Phi_{H_1}(t)$$

$$h(i) = h(i-1)\Phi_{H_i}(t), \text{ where } 2\le i\le M$$

Now, substituting for $\Phi H_i(t)$ we have:

$$h(i) = h(i-1)\frac{1}{1-X_it}$$

$$= h(i-1)(1+X_it++X_i^2t^2+\cdots$$

$$= h(i-1)+X_i h(i-1)$$

Let g(n,i) be the coefficient of the t^n term of the h(i) polynomial. Then, from the last equation we must have:

$$g(n,i) = g(n,i-1)+X_i g(n-1,i)$$

This is equivalent to Buzen's result for calculating the G(N), where G(N) = g(N,M).

This result is true if the ith server is load independent. What if the processing rate $\mu(j)$ is a function of the number of customers in the system? This is usually done to simulate parallel servers at a station or, more generally, some interaction between service rate and load (e.g., memory contention). As we have seen the probability distribution still has a product form solution. To calculate g(n,i) in this case we must essentially do a formal convolution and we have:

$$g(n,i) = \sum_{k=0}^{n} F_i(n-k)g(k,i-1)$$

where $F_i(k)=(X_i/\mu_i)^k/D_i(k)$ *as defined in Gordon and Newell*

Bruell [Bruell 80] gives a more efficient algorithm in the load dependent case where there are parallel servers. The number of operations required to solve G(N) for a network of M load independent stations with N customers is O(NM) and for a network of load dependent stations it is $O(NM^2)$. This is a substantial improvement over a straight summation of all states.

We have essentially given the computational algorithm that Buzen derived for calculating G(N). We have allowed for our servers to have exponential service time distributions and either the service rate is load independent or load dependent. Having calculated G(N), Buzen then went on to show that most of the important performance measurements can be calculated in terms of G(N). Let us just present his results:

Throughput: $T_i = e_i \dfrac{G(n-1)}{G(N)}$

Utilization: $U_i = X_i T_i$ *(load independent)*

$$U_i = 1-g^i\frac{(N,M)}{G(N)} \quad \textit{(load dependent)}$$

where $g^i(N,M)$ is the calculation of $G(N)$ with the ith station removed

Mean queue length: $N_i = \sum_{n=1}^{N} X_i^n \frac{G(N-n)}{G(N)}$ *(load independent)*

$$N_i = \frac{1}{G(N)} \sum_{n=1}^{N} nF_M(n)g(N-n,M-1) \quad \textit{(load dependent)}$$

where we assume that the load dependent station is the last station or Mth station

Average waiting time: $W_i = \frac{N_i}{T_i}$ *(Little's Law)*

The central server model assumes that a CPU is a central server. From the CPU, transitions are allowed to a number of servers representing I/O devices. Once the job has been processed by the CPU it is then serviced by one of the I/O devices (representing a disk, drum, or tape drive, for example). When an I/O service has been completed the program returns to the CPU queue. Thus, in the central server model program behavior is modeled as CPU processing alternating with I/O processing. This is thought to represent a multiprogrammed computer where many jobs reside in core competing for system resources. Buzen's results can easily be applied to the simple network implied by the central server model to obtain all of the performance measurements presented above. The central server model has proved to be an accurate predictor of system performance for batch multiprogrammed systems. It is also possible to extend the model for time-sharing systems.

12.5. Buzen Extended

The principal defects in queueing networks that can be solved by Buzen's techniques are : (1) all the customers are identical; they have the same service time distribution at each station (2) all the service time distributions are exponential and FCFS.

Experimenters, however, found the central server model to be an excellent predictor of performance even though the assumptions of the model appear to be violated. For example, the CPU service time is poorly approximated by an exponential distribution and is generally considered to be hyperexponential. Baskett

[Baskett 72] was able to explain this anomaly for the central server model. Up till now, the assumption has been that the queueing discipline is FCFS (first-come-first-served) for all stations. However, a CPU typically schedules jobs in a round-robin discipline. If we assume that the PS (processor sharing) discipline, which is the limit of the round-robin discipline as the time quanta approaches zero, is a good approximation of a processor rapidly switching between jobs, then the results can be explained. Baskett was able to show that a PS discipline with a service time distribution that can be described by a series of exponential stages (general Erlangian distribution) is equivalent to a FCFS single server with the same mean service time. This would then explain why the CPU in the central server model was insensitive to non-exponential service time distributions.

Chandy [Chandy 72] extended this result to closed Jackson networks where the service time distributions could be any that had a rational Laplace transform as long as the queueing discipline was PS or even LCFSPR (last-come-first-served-preemptive-resume). LCFSPR describes a queueing discipline in which a customer receiving service is replaced by a new customer entering the queue. The preempted customer's service is eventually resumed at the point at which it lost its service. This discipline could be used to model a CPU which is continually interrupted and must run at a higher priority than it is currently at. The original process will be resumed when the higher priority process completes. Finally, Muntz [Muntz 73] showed other situations where a queue would have a Poisson departure process given that the arrival process was Poisson. Muntz refers to this property of queues as Markov to Markov or M=>M. This is an important property, since as suggested by Burke's result, it leads to a product form solution for an open network. Specifically, Muntz shows that a queueing model with different classes of customers, each with their own service time requirements, has the M=>M property. This is under the assumption of a service distribution with a rational Laplace transform and a PS queueing discipline. Muntz shows that the M=>M property allows the equilibrium state probability of open and closed Jackson networks to be written in the product form.

Finally, the above results were formalized for open, closed and mixed networks in the BCMP theorem [Baskett 75]. In this model the product form solution is given for a Jackson network of different classes of customers. A class of

customers has defined for it a separate service time distribution for each station and a separate set of probabilities governing the movement of that class between stations. The model allows for a mixed network where it is open with respect to some classes and closed with respect to other classes. The BCMP results tell us that a network may consist of an arbitrary mixture of the following stations:

1. The service discipline is FCFS; all customers have the same service time distribution at this service center. The service rate can be some arbitrary function of the number of customers in the system.
2. There is a service center with a PS sharing queueing discipline and each class of customers may have a distinct service time distribution represented by a rational Laplace transform.
3. There is a service center with more servers than the number of customers in the network. This is known as an IS (infinite server) station. Each class of customers can have distinct service time distributions having a rational Laplace transform.
4. There is a single server with a last-come-first-served-preemptive-resume discipline (LCFSPR). Each class of customers may have a distinct service time discipline which may be described by a rational Laplace transform.

The multiple class networks allows for separate service rates and transition probabilities for each class. One must now solve a set of linear equations for each class of customers, just as was done in the single class model, to determine the network arrival rates on a class basis. The model also allows for transitions from one class to another and in this case one speaks of essential classes. A class of customers is allowed to switch classes at a station and there is a probability assigned to this transition. All the classes associated with a *chain* of class changes define an essential class. (The chain must be irreducible, which requires that every class within the essential class can be reached from any other class). Shum shows that a model of M classes where there are r <= M essential classes is equivalent to a non-class switching model which has r irreducible classes. This is an important result in general and allows a seemingly complicated model to be reduced to a simpler model with less classes and no switching.

What was provided by the BCMP theorem was a solution technique, rather than a closed form solution to the problem. However, because the solution has the

product form, one can extend the Buzen convolution algorithm to several dimensions. For a more complete discussion of the multi-dimensional computational queueing algorithms, see Bruell [Bruell 80] and Kobayashi [Kobayashi 75].

We are interested in the above results because it gives us more flexibility in modeling a computer system. Specifically, the PS discipline allows us to model a device which is known to be involved in several activities merely by determining what the average service time of the distribution represented by these activities is. Thus, for our message based system we know that for the workstations these activities include computation and copying of messages. Presumably, the average size of the segments that are read or written to the file server differ. That is, segments that are read from the server have a different mean size than those that are written. If we know the probability in which read and write request are generated then we are trying to determine the mean service time of a station whose distribution can be described by a network of exponential stages. This mean can be easily determined (see Kleinrock [Kleinrock 75]) and it is this mean service time we use in our computations. This idea carries through to all the devices we are using (Ethernet, file station, and disk).

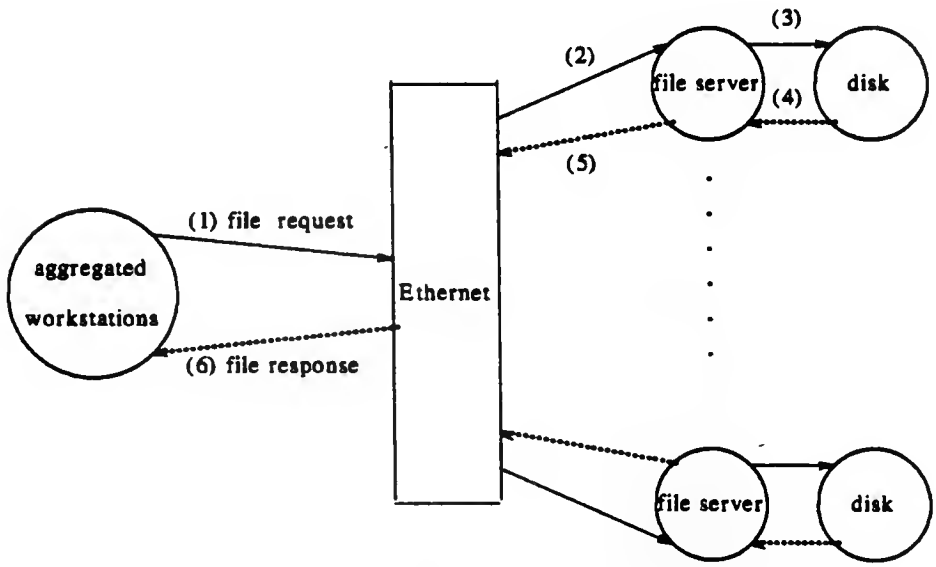
13. Analyzing the Distributed UNIX System

In an analysis of a distributed UNIX system, we are primarily interested in the performance of the remote file server in transferring large segments. The UNIX kernel, as mentioned previously, is small and efficient. It requires a program to be loaded for most commands entered from the terminal. Thus, most of the large segments read are programs being loaded from the file server and executed on the workstation. We can assume, for purposes of the model, that most writes to the file server are also segments.

In modeling the V file protocol, a reasonable first attempt at determining the performance is to model a problem-oriented protocol. As was mentioned previously, there is evidence to indicate that V does perform as well as LOCUS in this regard. Also, a problem-oriented protocol is easier to analyze with the techniques from the previous section and would indicate what the lower bounds of V could be. So, for the purposes of the model, a read request would consist of the request message followed by some file server activity and completed by the actual

delivery of the file pages. A write request is similar. This implies that we are not modeling exactly the paradigm of **send-receive-movefrom-reply**, but an approximation of this activity. However, as we will see, the model does account for any concurrency that the model might afford. The model will consist of the following stations:

1. an aggregation of N workstations each consisting of a 10 Mhz. Motorola 68000 processor as one load dependent PS service station.
2. a PS service station to approximate the 3 Mb. Ethernet connection.
3. a PS service station to represent the file processor (10 Mhz. Motorola 68000).
4. a PS service station to represent the disk with one disk per file processor



Network Model

We assume that all N workstation are identical and have the same behavior. Thus, we can aggregate them as one load dependent service station. To further test this assumption, I modeled a 2 class network consisting of an aggregation of N-1 workstations as one class and one lone work station to represent the other class. It

was then seen that the single workstation contributed 1/N to all the performance measurement (see Appendix C). Assuming that Ethernet behaves like a PS station is open to some criticism. However, Goldberg [Goldberg 83] makes the same assumption in his LOCUS model. Finally, we will model the remote file stations as consisting of one processor with a disk unit. The disk is assumed to be a PS station. The usual assumption for a disk is that it is a FCFS station. The PS station allows us to handle the situation where read and write requests have two different service times and thus the distribution is non-exponential (actually, hyperexponential). Although the actual mechanism for UNIX disk I/O is more robust than FCFS, Goldberg models this as FCFS.

Let us now examine how an I/O read request is treated in the model. The workstation is initially executing some program. With some probability a read request is generated. This read request is copied from user space into the UNIX kernel. It is then copied from the kernel space into the Ethernet interface. (Cheriton notes that overhead in network communication is not in the time for transmission but in the number of times a message gets copied.) The request is then transmitted over the Ethernet where it is received by the file server. The file server then copies the request from the DMA interface and into the kernel and from there into the space of the file server process. The file server process does some CPU processing and then it sends the request over to the disk to initiate a transfer. After the pages are received from the disk, the file server does some more CPU processing and then transmits the pages over the Ethernet and back to the workstation. They are copied out of the interface and into kernel space and from there back into the user space of the requesting process. The program then continues executing until the next program is execed or another I/O request is generated. A write request would be treated similarly.

The model requires a class change to allow for a request to revisit a station with a different mean service time. For example, a page write request leaves the file server for the disk. After the pages are written, the request returns to the file server. Normally, this would mean that a set of linear equations would have to be solved for each class. However, we are fortunate in that we really have only one essential class which is equivalent to a single class model. The class changes are artificial and we are really dealing with stations which are non-exponential (see

Sauer [Sauer 81] on this point). Since the service stations are PS, we are allowed these non-exponential distributions and are only interested in the mean service time.

Basic model constants (all times in milliseconds)	
Packet size	512 bytes
Processor copy time of packet	.4
Ethernet transmission time of packet	1.39
File server time	5.0
Read segment size (packets)	100
Write segment size (packets)	50
Read/Write reply size (packets)	1
Disk access time	15
Work station computation time	500

The table above gives the basic constants of the model. An important parameter is the time to copy a 512 byte packet. A remote transfer of a packet will involve 4 copies, a copy from user space into kernel space and from kernel space into the Ethernet interface, in each direction. An extra copy must be considered when pages are read from or written to the disk (from kernel space into user space). The file server time parameter is based on the time of the Locus file server (2.5 ms.) plus the processor time for a receive-reply (approximately 2.5 ms.). The disk access time of 15 ms. is assumed to be that of a large, fast disk. We assume that a read request message or a write reply message is a 512 byte packet. The size of file pages transferred between the disk and the workstation are dominated by the size of programs to be executed, assumed to be about 100 512-byte pages. A write to a file system is is also assumed to be a segment, rather than a random page write. However, its mean size is considerably smaller (50 pages). The LOCUS model also assumes that for each file page read, a workstation spends 5 ms. of processor time involved in some computation. We are essentially modeling the behavior of a

system that is under a continual load from the workstations. We are making no allowance for user *think time*. This is considered to be the mean time between commands typed on a terminal. This model should then be interpreted in this light. That is, we are essentially considering a situation where each user is compiling a set of programs (e.g., using the UNIX *make* command). A UNIX compile involves the execing of the UNIX compiler from the disk, the compilation of assembler code which is written to the disk, the execing of the assembler off of the disk, and finally the assembled code written to the disk. This activity makes heavy use of the file servers and would give us an indication of how well the system behaves during this period.

For each model to be computed, the following parameters are required:

1. the number of workstations.
2. the level of multiprogramming.
3. the number of file stations.
4. the probabilities of an I/O request going to each file station.
5. the probability that an I/O request is a read or write.

For all of the models considered in the following sections, the level of muli-programming is one. That is, the number of I/O requests outstanding in the system is equal to the number of workstations. The model also assumes that one disk is associated per file station. So, to simulate a two disk network, the model requires two file servers and the associated transition probabilities to these two file servers. The model also assumes that all file stations and disks are identical.

14. Results of One File Server

In Appendix B the performance measurements for 5, 10, and 15 workstations supported by one remote file server are given. Cheriton [Cheriton 83] estimates that one file server can support 10 workstations satisfactorily and up to 30 work stations with some delay. However, this was based on the important assumption that 90% of all file activity is for page requests. In the UNIX environment, most of the file activity is loading of programs off of the disk. We assume for purposes of the model that 70% of all file requests are reads of 100 file pages and 30% are writes of 50 file pages. In terms of the model, since all the stations are PS, this gives us an average segment size of 85 pages. In the LOCUS model, the average size is assumed to be about 100 pages.

The model indicates, as expected, that the workstations, Ethernet, and the file station are underutilized. The bottleneck in the model becomes our fast disk. This too could have been predicted since the disk service time per request is an order of magnitude greater than any other station's service time. We should expect that the disk waiting time will dominate our response time in the system. It should be noted that we have assumed that every request will force a disk access and we have not allowed for caching of recently used disk pages. We are really getting a worst case analysis of the model. But, having considered this, the performance of the system does justify our use of remote file servers. If we are transferring on the average 85 disk pages per request at 15 ms. of disk access time per page, then a waiting time of at least 1.25 sec. per request can be expected. If there are 15 users in the system, then we might expect a waiting time of approximately 19 seconds assuming a highly utilized disk. This compares with a waiting time of 18.3 seconds at the disk and a system response time of 18.4 based on the results of the model (see Appendix B). This argues that our system is competitive with a traditional time-sharing system in terms of file system performance and probably in terms of cost. This, also, does not take into account that our workstations and Ethernet are underutilized. We could run a program that is I/O bound in the background while running a program that makes use of the ipc. facilities to communicate with a server on another processor and expect no significant increase in our response time.

It should be noted that the response time given in Appendix B is the average time a request spends in the queueing network outside the workstations. We can calculate this exactly, since with only 1 request outstanding per workstation, there is never a queueing delay at our load dependent station representing our aggregated workstations. A request is immediately serviced in our model. (Essentially, the load dependent workstation is an IS (infinite server) station). Thus, we can subtract the average queueing length of the workstation from the total number of requests in the model to get the average queueing length in the remainder of the system. By applying Little's Law to this, we obtain the average response time (see Bruell [Bruell 80] on this point).

15. Results of Two File Servers

The results from the one file server model with 15 workstations indicate that a second file station might be required to get satisfactory performance. The results of the performance of this model are given at the end of Appendix B. This model assumes that file requests can be equally divided between each file station. While this might not be obtainable in practice, it does give us optimal response time (assuming the disks are identical). As we can see, this response time is practically half of that of the one server model giving us a response time of 9.8 seconds. This I believe to be an acceptable response time for a heavily loaded 15 work station configuration and seems to compare favorably with response time for the LOCUS model. LOCUS has a similar architecture to the V kernel where processor sites are connected by a faster 10 Mb. Ethernet. A LOCUS site is a VAX 11/750 and includes disks (at least 2 per processor). Goldberg's model indicates that LOCUS will give a response time of approximately 6-7 seconds for 15 users per site in a 4 site model where 80% of the file requests are processed *locally*. The remaining 20% of the file requests are processed by remote sites. The results of Goldberg's model in this case has been validated experimentally. Since most of my important parameters are based on the Goldberg model of LOCUS, the 9.9 second response time for entirely remote file transactions seems to indicate that the model is behaving well. That is, the LOCUS model in this case is practically a traditional central server model (80% local file requests make it this) and we would hope that a 15 workstation model would give similar results to a single processor system. Goldberg also presents a response time for LOCUS derived experimentally of approximately 8 seconds for a 4 site 24 user system with 100% remote file access. This compares with 7.8 seconds response time for a 4 file server model with 22 users for our model (see Appendix B). This would seem to indicate that our model is predicting accurately a similar LOCUS configuration. And, since we have been attempting to show that a V-like system and LOCUS have similar performance, this is strong evidence for this view. However, it should be added that I have indeed been modeling the LOCUS problem oriented protocol. The actual V protocol would involve the sending of two more messages (the send and reply messages). However, the model does show that Ethernet is very much underutilized and that the overhead in sending these messages in this case would be small.

It can be concluded from this that to support 15-20 workstations satisfactorily in the UNIX environment then 2 file servers are required. This perhaps is not surprising, although Cheriton [Cheriton 83] seems to suggest that one file server is adequate in this situation. The analysis of V's page level file access indicates there is the possibility of getting better performance than a local file system with a slower disk . However, this assumes that *no one else* is using the system. In the distributed UNIX environment that is being proposed, with many users attempting to load programs from the disk and over the network, we seem to be getting comparable performance to a traditional time-sharing single processor system. This is good news since there are advantages to this system. The local UNIX kernels running on the workstations are small (they have no file system code) and cheap (they have no disks). Of course, one can also write distributed programs.

There are, however, optimizations to be made. For one, certain programs that might expect to be loaded frequently can be kept locally (e.g., the UNIX shell or the C compiler) in ROM memory perhaps. The model indicates that the file station processors are underutilized. It would make sense to have I/O intensive file operations (e.g., copying of one file to another) performed remotely rather then locally. Finally, the model results indicate that a software cache maintained by the remote file server could make a dramatic difference in response time for a heavily loaded system.

16. Conclusion

I believe it can be concluded that it is possible to build a distributed UNIX system with diskless workstations using remote file servers and still have comparable performance to a traditional time-sharing system. The results seem to indicate that 2 file servers are required for every 15 - 20 users to provide adequate performance. Although a problem oriented protocol was modeled, the V protocol should not add significantly more to the response time as long as Ethernet remains underutilized. The bottleneck at the disks indicate that the best way to improve response time is through a software cache which would allow read-ahead and write-behind of file pages. UNIX does already implement such a cache for its file system. So, we could expect an actual system to perform even better than the model indicates.

The model validates a method of message passing which strives for simplicity and efficiency. Although, we did not model exactly the V file access protocol, the 2 extra messages which would be sent in practice should not add significant processor or network overhead (especially since the model indicates these resources are underutilized). Thus, we can expect similar response time to the model. Finally, I think this thesis has shown a way to extend UNIX for a distributed environment which is consistent with UNIX and which would perform well.

Appendix A.

```
/*
  message buffer and header for proc structure
*/
#define munixsize 32 /*size of message buffer in bytes*/

struct msg_header {
   short m_id;
   int mtime;
};
struct msg_buffer {
   struct msg_header m_header;
   char m_data[munixsize];
}

#define Mgh  msg_buffer.m_header
#define Mgd  msg_buffer.m_data

#ifdef SMALL
#define SQSIZE 010
#else
#define SQSIZE 0100
#endif

#define SMSG 7  /*new state for a process: waiting on a message queue*/
#define PMSG (PZERO+1) /* when sleeping (reply blocked) make him interuptible
                                                so he can be freed in case of problems*/

#define EPID 67   /*invalid pid*/
#define EMIO 68   /*message error*/
```

```
/*
 HASH function used in Berkeley UNIX to determine
 which queue a proc entry is on
*/
#define HASH(x) ( (int) x >> 5) & (SQSIZE-1))

/*
  states a process can be in
  when using message primitives
*/
#define SNDB 010000      /*waiting for a receiver*/
#define RECVB 020000 /*waiting for a sender*/
#define RPLB 040000      /*waiting for a reply*/
#define RCVIB 0100000 /*waiting for a specific sender*/

/*
  new queue structure for UNIX: message queue
  senders queue themselves for corresponding receivers
*/

struct proc *msgque[SQSIZE];

extern struct proc *sipque[];
extern struc proc *maxproc;
```

```
/*
  send primitive
*/
send()
{
 register struct proc *p, *q;
 register caddr_t ml;
 register char c;
 short pid;
 int h,s;

 p=u.u_procp; /*establish proc entry*/
 pid = u.u_r.r_val2;

 if(!valid_pid(pid)) {/*search proc table to check existence of receiver*/
  u.u_error=EPID;
  return;
 }

 /*set up for a write from user space into kernel space*
 u.u_base=u.u_r.r_val1;
 u.u_count=munixsize;
 u.u_segflg=0;

 h=HASH(msggque+HASH(pid)); /obtain index into sleepq*/
 p->Mgh.m_id=pid;
 s=spl6();

 for(q=slpque[h]; q; q=q->p_link) {

  if(q->p_pid != pid)  /*potential receiver */
   continue;
```

```
if(q->p_flag&RCVB || (q->p_flag&RCVIB && q->Mgh.m_id==p->p_pid))
  break; /* and is receive blocked or receive-specific blocked*/

}

if(q=NULL) { /*put proc on message queue; waiting for receiver*/

 /*copy message into buffer*/
 splx(s);
 ml=p->Mgd;
 while( (c=cpass()) != -1)
   *ml++ =c;

 if(u.u_error)
   return;

 setmq(p,pid);   /*put on message queue*/

 msleep();       /*give up processor*/

 if(p->Mgh.m_id==-1)
   u.u_error=EMIO;
 else {  /*got a reply*/
   /*copy out reply*/
   u.u_base=u.u_r.r_val1;
   u.u_count=munixsize;
   u.u_segflg=0;
   ml=p->Mgd;
   while(passc(*ml++) != -1);
 }

 u.u_r.r_val1=p->Mgh.m_id;
 return;
```

```
  }
else { /*found a receiver*/

  /*copy into receiver's message buffer*/
  ml=q->Mgd;
  while( (c=cpass()) != -1)
    *ml++=c;

  if(u.u_error) {
    splx(s);
    return;
  }

  /*receiver is no longer receive blocked*/
  q->p_flag &= ~(RCVB | RCVIB);
  q->Mgh.m_id=p->p_pid;
  setrun(q); /*wakeup receiver*/
  splx(s);

  /*waiting for a reply*/
  /*sender is now reply blocked*/
  p->p_flag |= RPLB;
  sleep( caddr_t)(msgque+HASH(pid)),PMSG);

  if(p->Mgh.m_id==-1)
    u.u_error=EMIO;
  else {
    /*set up for a copy out*/
    ml=p->Mgd;
    u.u_base=u.u_r.r_val1;
    u.u_count=munixsize;
    u.u_segflg=0;
```

```
    while(passc(*ml++) != -1);
  }

  u.u_r.r_val1 = p->Mgh.m_id;
 }
}
```

```
rcv(pid)
short pid
{
 register struct proc *mp,*lp, **t;
 struct proc *h, *p;
 register caddr_t ml;

 int h,g,s;

 p=u.u_procp;

 t=msgque+HASH(p->p_pid);  /*hash on a unique address*/

 /* is there a sender waiting for me*/

 s=spl6();

 if(*t) { *someone is there*/
   lp=*t; /*tail pointer*/
   hp=mp= (*t)->p_link; /*head pointer*/

   do {
     if(mp->Mgh.m_id==p->p_id) { /*found a sender*/
       if((pid != mp->p_pid) && pid) /*receive specific*/
         goto nxt;

       /*set up for a copy out*/
       ml=mp->Mgd;
       u.u_base=u.u_r.rval1;
       u.u_count=munixsize;
       u.u_segflg=0;

       while(passc(*ml++)) != -1);
```

```
    if(u.u_error) {
      splx(s);
      return;
    }

    if(*t == mp) /*remove tail*/
      if((*t=lp)== mp)
        *t=NULL;

    lp->p_link=mp->p_link;

    /*sender is now reply blocked; put him on sleep queue*/
    mp->p_flag &= ~SNDB;
    mp->p_flag |= RPLB;


    putasleep(mp,(caddr_t)(msgque+HASH(p->pid)));
    u.u_r.r_val1=mp->p_pid;
    splx(s);
    return;
  }

nxt:
  lp=mp;
  mp=p->p_link;
  } while (mp != hp);

 /*block the  receiver*/
 splx(s)
 p->p_flag |= (pid?RCVIB:RCVB);
 p->Mgh.m_id=pid;
 sleep((caddr_t)msgque+HASH(p->p_pid)),PMSG);
 ml=p->Mgd;
```

```
/*receive complete*/
if(p->Mgh.m_id==-1) {
  u.u_error=EMIO;
  return;
}
else { /*copy out*/
  u.u_base=u.u_r.r_val1;
  u.u_count=munixsize;
  u.u_segflg=0;
  while(passc(*m1++) != -1);
  u.u_r.r_val1= p->Mgh.m_id;
 }
}
```

```
/*
 receive specific
*/

receivid()
{
 short pid;

 pid = u.u_r.rval2;

 if(!valid_pid(pid)) {
    u.u_error = EPID;
    return;
 }

 rcv(pid);
}

/*
  receive general
*/

receive()
{
 rcv(0);
}
```

```
/*
 reply
*/

reply()
{
 register caddr_t ml;
 register struct proc *p,*q;
 short pid;
 int g,h,s;
 char c;

 p=u.u_procp;
 pid=u.u_r.r_val2;

 if(!valid_pid(pid))
   u.u_error=EPID;
   return;
 }

 /*set up for a copy into senders message buffer*/
 u.u_base=u_r.rval1;
 u.u_count=munixsize;
 u.u_segflg=0;

 u.u_r.r_val1=-1; /*if pid not found*/
 s=spl6();
 for(q=spique[h];q;q=q->p_link) {
   if(q->p_pid != pid)
     continue;

   if((q->p_flag&RPLB)  && (q->Mgh.m_ld==p->p_pid)) {
     ml=q->Mgd;
```

```
    while((c=cpass()) != -1)
      *ml++=c;

    if(u.u_error) {
      splx(s);
      return;
    }

    q->p_flag &= ~RPLB;
    setrun(q);

    splx(s);
    u.u_r.r_val1=pid;
    return;
  }
 }
splx(s);

}
```

```
/*
 put a proc entry on the special message queue
*/

setmq(p,pid)
register struct proc *p;
short pid;
{
 register struct proc **t;
 int s,h;

 s=spl6();
 t=msgque+HASH(pid);

 if(*t) { /* something there
   p->p_link==(*t)->p_link;
   (*t)->p_link=p;
 }
 else
   p->p_link=p;

 *t=p;

 p->p_flag |= SNDB; /*sender is send blocked*/
 p->p_stat=SMSG;  /*state is waiting on a message queue*/
 p->p_pri=PZERO; /*do not disturb*/

 p->p_wchan=0;
 p->Mgh.mtime=MTIMO; /* time out value, NOTE: clock will wake him up after*/
                /*  time is up*/
 splx(s);
}
```

```
/*
 put a proc (not yourself) on a sleep queue*/

putasleep(p,a)
register struct proc *p
caddr_t a
{
 register struct proc **hp;

 s=spl6();
 hp=&sleepque[HASH(a)];

 p->p_link= *hp;
 p->p_stat=SSLEEP;
 p->p_wchan=a;
 p->p_pri=PMSG;
 *hp=h;
 splx(s);
}

/*
  do a sleep; don't check for signals while on message queue
  only way to be woken up on message queue prematurely is through
  a timeout; however, when reply blocked can be signalled
*/
 msleep()
 {
  swtch(); /*do a UNIX swtch*/

  if(ISSIG(u.u_procp)) /*if signalled non-local goto to return from syscall*/
    resume(u.u_procp->p_addr,u.u_qsav);

 }
```

```
valid_pid(pid)
short pid;
{
 register struct proc *p;

 for(p=&proc[0];p <= maxproc; p++)
   if(p->p_pid=pid)
     return(1);

 return(0);
}
```

Appendix B.

Number of workstations:5

Number of file stations:1

Probability of a read request: .7

Probability of a write request: .3

.....

Throughput

Station	Throughput (requests/sec)
Work station	0.784
Network	0.784
File Ser: 1	0.784
Disk : 1	0.784

.....

Utilization

Station	Utilization
Work station	35.983%
Network	9.369%
File Ser: 1	8.387%
Disk: 1	99.934%

.....

Station	Aver. Qu. Length	Average Waiting Time (seconds)
Work Station	0.446	0.569
Network	0.103	0.132
File ser:1	0.091	0.117
Disk: 1	4.359	5.562

.....

Response time: 5.654

Number of workstations:10

Number of file stations:1

Probability of a read request: .7

Probability of a write request: .3

.....

Throughput

Station	Throughput (requests/sec)
Work station	0.784
Network	0.784
File Ser: 1	0.784
Disk : 1	0.784

.....

Utilization

Station	Utilization
Work station	35.989%
Network	9.376%
File Ser: 1	8.392%
Disk: 1	100.000%

.....

Station	Aver. Qu. Length	Average Waiting Time (seconds)
Work Station	0.446	0.569
Network	0.103	0.132
File ser:1	0.092	0.117
Disk: 1	9.359	11.932

.....

Response time: 12.025

Number of workstations:15

Number of file stations:1

Probability of a read request: .7

Probability of a write request: .3

.....

Throughput

Station	Throughput (requests/sec)
Work station	0.784
Network	0.784
File Ser: 1	0.784
Disk : 1	0.784

.....

Utilization

Station	Utilization
Work station	35.989%
Network	9.376%
File Ser: 1	8.392%
Disk: 1	100.000%

.....

Station	Aver. Qu. Length	Average Waiting Time (seconds)
Work Station	0.446	0.569
Network	0.103	0.132
File ser:1	0.092	0.117
Disk: 1	14.359	18.307

.....

Response time: 18.400

Number of workstations:15

Number of file stations:2

Probability of a read request: .7

Probability of a write request: .3

Probability of file station 1 request: .5

Probability of file station 2 request: .5

.....

Throughput

Station	Throughput (requests/sec)
Work station	1.462
Network	1.462
File Ser: 1	0.731
Disk : 1	0.731
File Ser: 2	0.731
Disk : 2	0.731

.....

Utilization

Station	Utilization
Work station	56.538%
Network	17.475%
File Ser: 1	7.821%
Disk: 1	93.194%
File Ser: 2	7.821%
Disk: 2	93.194%

.....

Station	Aver. Qu. Length	Average Waiting Time (seconds)
Work Station	0.832	0.569
Network	0.211	0.145
File ser:1	0.085	0.116
Disk: 1	6.894	9.431
File ser:2	0.085	0.116

Disk: 2 6.894 9.431

.....

Response time: 9.872

Number of workstations:22

Number of file stations:4

Probability of a read request: .7

Probability of a write request: .3

Probability of file station 1 request: .25

Probability of file station 2 request: .25

Probability of file station 3 request: .25

Probability of file station 3 request: .25

.....

Station	Aver. Qu. Length	Average Waiting Time (seconds)
Work Station	1.547	0.569
Network	0.479	0.176
File ser:1	0.078	0.115
Disk: 1	4.915	7.229
File ser:2	0.078	0.115
Disk: 2	4.915	7.229
File ser:3	0.078	0.115
Disk: 3	4.915	7.229
File ser:4	0.078	0.115
Disk: 4	4.915	7.229

.....

Response time: 7.880

Appendix C

Number of workstations:15

Number of file stations:1

Probability of a read request: .7

Probability of a write request: .3

Class 1 is a single workstation

Class 2 is an aggregation of 14 workstations

Throughput

Station	Job Class	Throughput(requests/sec)
Work station	1	0.052
Work station	2	0.732
Network	1	0.052
Network	2	0.732
File Ser: 1	1	0.052
File Ser: 1	2	0.732
Disk : 1	1	0.052
Disk : 1	2	0.732

.....

Utilization

Station	Job Class	Utilization
Work station	1	2.974%
Network	1	0.625%
Network	2	8.751%
File Ser: 1	1	0.559%
File Ser: 1	2	7.833%
Disk: 1	1	6.667%
Disk: 1	2	93.333%

.....

Station	Class	Queue Length	Waiting Time
Work station	1	0.030	0.569
Network	1	0.007	0.132

Network	2	0.097	0.132
File ser:1	1	0.006	0.117
File ser:1	2	0.086	0.117
Disk :1	1	0.957	18.307
Disk :1	2	13.402	18.307

.....

Response time for class 1: 18.556

References

[Bartlett 78] Bartlett, J.F., "A NonStop Operating System," *Eleventh Hawaii International Conference on System Sciences*, January 1978, pp. 103-117.

[Bartlett 81] Bartlett,J.F., "A NonStop Kernel", *Proceedings of the Eighth Symposium on Operating Systems Principles*, December 1981, pp. 22-29.

[Baskett 75] Baskett, F. et al.,"Open, Closed and Mixed Networks of Queues with Difference Classes of Customers," *J. ACM*,22(2), April 1975,pp. 248-260.

[Baskett 78] Baskett,F. et. al., "Task Communication in Demos," *Proceedings of the Sixth Symposium on Operating Systems Principles*, ACM, November 1977, pp. 23-31.

[Bechtolsheim 82] Bechtolsheim, A. et. al., "The SUN Workstation Architecture," tech. report, Computer Science Department, Stanford, Calif., Jan. 1982.

[Blair 83] Blair, Gordons S. et al., "A Practical Extension to UNIX for Interprocess Communication,"*Software Practice and Experience*, 13(1), 1983, pp.45-58.

[Bruell 80] Bruell, Steven C. and Gianfranco Balbo, *Computational Algorithms for Closed Queueuing Networks*, Elsevier North-Holland, New York, 1980.

[Buhr 84] Buhr, R.J.A, *System Design with Ada*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1984.

[Burke 56] Burke, P.J., "The Output of a Queueing System," *Operations Research*, 4, 1956, pp. 699-704.

[Buzen 71] Buzen, J.P., "Queueing Network Models of Multiprogramming", Ph.d Thesis, Division of Engineering and Applied Physics. Harvard U., Cambridge, Mass., May 1971.

[Buzen 73] Buzen, J.P., "Computational Algorithms for Closed Queueing Networks with Exponential Servers", *CACM*, 16(9),Sept 1973, pp. 527-531.

[Chandy 72] Chandy, K.M., "The Analysis and Solutions for General Queuing Networks," *Proceedings of the 6th Princeton Conference on Information Sciences and Systems*, Princeton, N.J., March 1972, pp. 224-228.

[Cheriton 79] Cheriton, M., "Thoth, A Portable Real-Time Operating System", *CACM*, 22(2), February 1979,pp. 105-115.

[Cheriton 81] Cheriton, M. , "Distributed I/O using an Object-based Protocol," Tech. Rept. 81-1, Computer Science, University of British Columbia, 1981.

[Cheriton 82] Cheriton, M., *The Thoth System, Multi-process Structuring and Portability*, American Elsevier, New York, 1982.

[Cheriton 83] Cheriton, M. and Willy Zwaenepoel,"The Distributed V Kernel and its Performance for Diskless Workstations", In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles,Operating Systems Review*, 17(5), 1983, pp. 129-140.

[Cheriton 84] Cheriton, David R.,"The V Kernel: A Software Base for Distributed Systems", *IEEE Software*, 1(2), October 1984, pp. 19-43.

[Cheriton 84b] Cheriton, David R., "An Experiment using Registers for Fast Message-Based Interprocess Communication," *Operating Systems Review*, 1984, pp. 12-20.

[Gentleman 81] Gentleman, W. ,"Message Passing Between Sequential Processes: the Reply Primitive and the Administrator Concept," *Software Practice and Experience*, 11(5), 1981, pp. 435-466.

[Goldberg 83] Goldberg, Arthur and Gerald Popek,"A Validated Distributed System Performance Model," *Performance '83*, North-Holland Publishing Co., 1983, pp. 251-268.

[Gordon 67] Gordon, W.J. and Newell, G.J., "Closed Queueing Systems with Exponential Servers," *Operations Research*, March 1967, pp 254-255.

[Gordon 67b] Gordon, W.J. and Newell, G.J., "Acknowledgement," *Operations Research*, December 1967, pp. 254-265.

[Hansen 78] Hansen, P. Brinch , "Distributed Processes: a concurrent programming concept," *CACM*, 21(11), pp. 934 -940.

[Jackson 57] Jackson, J.R.,"Networks of Waiting Lines," *Operations Research*, 1957, pp. 518-521.

[Jackson 63] Jackson, J.R., "Jobshop-Like Queuing Systems," *Management Science*,10(1), October 1963, pp. 131-142.

[Koenigsberg 58] Koenigsberg, E. "Cyclic Queues," *Operations Res. Quart.*, 9(1), 1958, pp. 22-35.

[Jalics 83] Jalics, Paul J. and Thomas S. Heines, "Transporting a Portable Operating System: UNIX to an IBM Minicomputer," *CACM*, 26(12), December 1983, pp. 1066-.

[Kleinrock 75] Kleinrock, L., *Queueing Systems, Volume I: Theory*, Wiley-Interscience, New York, 1975.

[Lions 77] Lions, J.,"A Commentary on the UNIX Operating System," Dept. of Computer Science, The University of New South Wales, 1977.

[Luderer 81] Luderer,G.W.R. et. al., " A Distributed UNIX System Based on a Virtual Circuit Switch," *Proceedings of the 8th Symposium on Operating Systems Principles*, ACM, December 1981, pp. 160-168.

[Metcalfe 76] Metcalfe, R.M. and D.R. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks," *CACM, 19(7), July 1976, pp. 395-404.*

[Lycklama 78] Lycklama, H. and D.L. Bayer,"The MERT Operating System," *Bell System Technical Journal*, 17(6), pp. 2049-2085, 1978.

[Nelson 1981] Nelson, B.J., "Remote Procedure Call," Phd. thesis, Carnegie-Mellon University, Pittsburg, Pa., 1981.

[Ousterhout 80] Ousterhout, J.K., D.A. Scelza and P.S. Sindhu, "Medusa: an experiment in distributed operating system structure," *CACM*, 23(1), 1980, pp. 92-105.

[Popek 81] Popek, G. et al.,"LOCUS: A Network Transparent, High Reliability Distributed System", *Proceedings of the 8th Symposium on Operating systems Principles*, ACM, December 1981,pp. 169-177.

[Muntz 73] Muntz, R.R., "Poisson Departure Process and Queueing Networks", *Proceedings of the 7th Annual Princeton Conference on Information Sciences and Systems*, Princeton University, Princeton, N.J., March 1973, pp. 435-440

[Rashid 81] Rashid, R. and G. Robertson, "Accent: A Communication-Oriented Network Operating System Kernel", *Proceedings of the 8th Symposium on Operating systems Principles*, ACM, December 1981, pp. 64-75.

[Reiser 74] Reiser, M. and H. Kobayashi, "Queueing Networks with Multiple Closed Chains: Theory and Computational Algorithms," *IBM Journal of*

Research and Development, 19, 1975, pp.283-294.

[Russel 77] Russel, D.L., "Process Backup in Producer-Consumer Systems", *Proceedings of the Sixth Symposium on Operating Systems Principles*, November 1977, pp. 151-157.

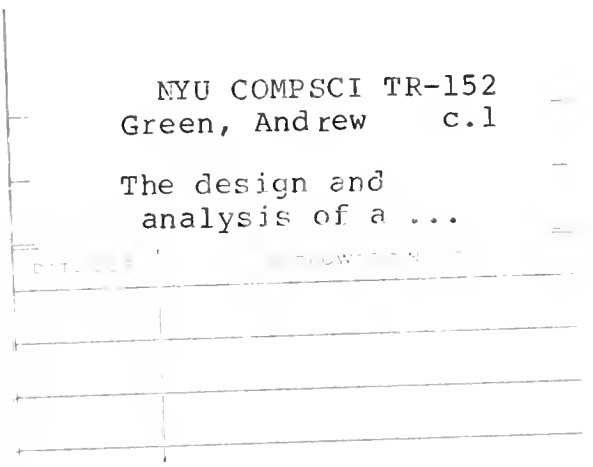
[Ritchie 74] Ritchie, D. and K. Thompson, "The UNIX Time-Sharing System",*CACM*, 17(7), July 1974,pp.365-375.

[Ritchie 78] Ritchie, D., "A Retrospective," *Bell System Technical Journal*, 57(6), 1978.

[Saltzer 80] Saltzer, J. et al., "End-to-end arguments in System Design", Notes from IEEE Workshop on Fundamental Issues in Distributed Systems, Pala Mesa, Ca., Dec. 15-17, 1980.

[Sauer 81] Sauer, Charles H. and K.M. Chandy, *Computer Systems Performance Modelling*, Prentice Hall, New Jersey, 1981.

[Shum 76] Shum, A.W.C., "Queueing Models for Computer Systems with General Service Time Distributions," Ph.d Thesis, Division Engrg. and Applied Physics, Harvard Univ, Cambridge, Mass., December 1976.



NYU COMPSCI TR-152
Green, Andrew c.1

The design and
analysis of a ...

N.Y.U. Courant Institute of
Mathematical Sciences
251 Mercer St.
New York, N. Y. 10012